Global Business Objects: Requirements and Solutions

Walter Bischofberger[1], Michael Guttman[2] and Dirk Riehle[1]

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

[2] Genesis Development Corporation
10 North Church Street, 4th Floor West Chester
PA 19380 USA

e-mail: {Walter.Bischofberger, Dirk.Riehle}@ubs.com, mguttman@gendev.com

Developing world-wide distributed object-oriented systems poses a number of difficult problems. In this paper, we summarize some of these problems as a set of requirements and we present our software architecture that addresses them. Our software architecture is reflective in all its key abstractions which is a prerequisite to successfully satisfy the requirements. Furthermore, it defines a set of common capabilities and standard implementations. Key capabilities comprise support for persistence, migration, event handling, transactions, hooking up inspection, debugging and security mechanisms. A mainstream banking application which conforms to the architecture is currently being developed at UBS, with Genesis as the lead consultant. Ubilab is focusing on the research aspects of the project such as type and domain model evolution, world-wide web integration, and introducing higher-level abstractions of software architecture which go beyond single classes and objects.

1 Introduction and Motivation[3]

Union Bank of Switzerland (UBS) is a large globally operating bank. Its operations require more and more world-wide distributed applications. New applications must integrate with old applications and must be prevented from turning into legacy applications themselves. To address these problems, we are working on a homogenous, reflective object-oriented software architecture. This architecture provides both a framework for developing new applications and for wrapping and integrating existing systems.

In this paper, we present some requirements for this architecture which we consider to be particularly important. These requirements are: support for evolution from the very first day, flexible use and integration of existing and new middleware, availability of generally useful capabilities to improve design and code reuse, and the possibility to introduce new software architecture abstractions which go beyond single classes and objects. We present a software architecture which fulfills these requirements, as we believe.

The architecture definition is centered around a distributed object-oriented virtual machine which provides a small number of reflective key abstractions. The virtual machine consists of a reflective kernel which provides the “primitives” of evolution. It allows to change, replace, and thereby evolve single type interfaces and implementations while providing several versions at the same time. Based on these primitives, we intend to support evolution on a larger scale: types usually do not evolve in isolation, but rather in groups of related interfaces and implementations. These groups of types can either be partial or full domain models or infrastructure frameworks together with all their clients. This must be supported by proper modeling notations and tools.

---

[3] In *Proceedings of the Ubilab Conference ’96, Zurich.* Universitätsverlag Konstanz, 1996. Pages 79-98.

If a system cannot be shutdown for maintenance purposes, it is difficult to install and replace type interfaces and type implementations. Unfortunately, this is the case with most non-trivial globally distributed systems. It is also very difficult to run integration tests for new or changed applications. Within our reflective architecture, we propose to treat types as first-class objects so that their installation, evolution and testing can be handled within regular transactions. This is a big advantage of our approach, something not yet offered by systems based solely on current industry standards like CORBA.

At first glance, building frameworks and business object models for large-scale distributed applications is not very different from the development of frameworks and business object models for single process applications. However, it is obvious that the complexity increases significantly because distribution requires developers to cope with all problems inherent in distributed applications. We therefore want to apply—and enhance—today's collective experience in developing frameworks [Lew95] to develop globally distributed systems.

It becomes considerably easier to develop frameworks if certain capabilities are guaranteed for all objects of any type. We have experienced this again and again in developing frameworks based on the ET++ [Wei94] infrastructure or Smalltalk [Gol89] in contrast to using pure C++ and its rudimentary libraries. Therefore, we decided to guarantee certain key capabilities for every object of any type. These key capabilities provide support for persistence, migration, event handling, transactions, hooking up of inspection, debugging and security mechanisms and others more. For all such key capabilities standard implementations have to be available which can be replaced easily if necessary.

Whenever we present our architecture we have to argue why we invest all this effort to solve problems that already seem to have been addressed by CORBA-compliant systems.[4, 5] We believe that current CORBA-based systems fall short in several aspects, in particular:

- Evolution support requires reflective capabilities like type system transactions that have not even been addressed by CORBA.
- Guaranteeing runtime behavior is almost impossible with CORBA, in particular since it is provides a rather vague notion of QOS (quality of service), which is supposed to distinguish implementations of different vendors. No semantics and runtime behavior specifications are available and can be utilized so that it is impossible to manage and therefore guarantee more than the most trivial aspects of runtime behavior.
- CORBA provides a broad range of useful services. Unfortunately, CORBA took the approach of C++ of not prescribing any kind of functionality to be provided by any object of any type. Thus, every project will both reinvent what is required by every object to work properly, as well as implement this anew. This limits design and implementation reuse.

---

4 In this paper, we focus our critiques of existing commercial object systems upon CORBA because it is the best known and most widely accepted standard for distributed object computing. However, the same critiques can be made of other commercialized object models, for example, Microsoft's OLE, the C++ object model, etc.

5 These discussions remind us very much of earlier—partially religious—discussions such as why the Macintosh is better than an IBM PC or why PL/1 was not the last programming language that was developed although it solved all problems

In short, the problem with CORBA is what it doesn't yet specify—a coherent set of models and policies for managing rapidly evolving, large-scale distributed systems.

UBS is undertaking this effort in form of an informal cooperation between GINS/GITA, a division of UBS, and Ubilab, the information technology laboratory of UBS. The authors of this paper are three of the four authors of the key software architecture specification document [Bis96]. As the lead consultant in the GINS/GITA project, Genesis Development Corporation has introduced the basic model of a reflective virtual object machine [Gen95] as a proposed element of the UBS Systems Architecture, developed based on Genesis' many years of experience in large-scale distributed object systems. Earlier this year, a prototype has shown the feasibility of the approach in a limited setting.

Ubilab is focusing on the research aspects of the project. We will work on evolution support, evaluate and devise frameworks, integrate world-wide web support into the architecture, and prepare the grounds for introducing higher-level abstractions of software architecture on the system and business domain level.

In Section 2 of this paper we review the requirements for a software architecture supporting global business objects. Section 3, 4, 5 and 6 first present and then discuss our solutions in form of a reflective, distributed, object-oriented virtual machine. In Section 7 we compare our work with other approaches. In Section 8 we draw our conclusions and outline how we will proceed further in this project.

2 Detailed Requirements

Developing world-wide distributed object systems poses a number of challenging requirements and problems, which we discuss in this section. The process of developing the architecture specification discussed in the next section was intertwined with our discussion of the requirements for it. Even though we had done some preceding analysis, the specification of the architecture could not have been separated from the analysis of the requirements. This process heavily drew on the experience of the involved people.

We identified the following main requirement categories:

- set of basic capabilities and services,
- support for large scale software development,
- introducing new software architecture abstractions,
- support for graceful evolution,
- reusability of standard implementations,
- integration with existing infrastructure.

We discuss the issues from a research perspective by detailing what is desirable, not necessarily by what is possible with today's concepts and techniques. An example is the need for object space transparent transactions, that is transactions which are not limited to database execution contexts. Generic transactions are a requirement, although, it is obviously a hard research problem to implement them reliably.

2.1 Set of basic capabilities and services

Every project and every application must provide certain *capabilities* and requires certain *services* to build upon. We define the notion of capability and service before we discuss the capabilities themselves. We clearly define the meaning of these words because we consider other definitions to be confusing, in particular those introduced by CORBA [Sie96].

A *capability* offered by an object is some functionality which makes it usable for specific clients within a domain model. It is expressed as an interface. Types the instances of which offer this capability inherit from this interface. Examples are the capability of an object to provide a passive data representation of itself or the capability to announce events about state changes. Usually, objects offer more than one capability. Capabilities represent a contract between the instances of a type and their clients. This functionality can be implemented in various ways. It is, for example, possible that for performance reasons objects of the same type use different implementations for the same capability.

A *service* provides some useful functionality to an unknown number of clients which are not confined to be part of a specific domain model. A service is expressed as an interface which clients directly use. Examples of services are naming and transaction management. An object providing a service usually focuses on this single service. A service is usually part of the underlying infrastructure. It has to implement its functionality but does not have to fulfill all the requirements of a domain object.

More pragmatically speaking, a capability is something rather self-contained within an object (which might be implemented with the help of some services), while a service is something which is understood as some kind of front-end to sometimes very elaborate implementations transcending the boundaries of the homogenous architecture and being intertwined with concrete middleware and systems software.

Some capabilities and services are business domain dependent, some are not. Not being part of a business domain means being part of the system domain, the major focus of this paper. Capabilities and services are the common foundation on top of which further frameworks can be built. They have therefore to be as homogeneous and orthogonal to each other as possible. There are many mandatory capabilities and services for any large distributed system. Some of them are discussed in the following list.

- *Naming and lookup of objects.* It must be possible to name objects in order to obtain references to them. Looking up the objects and potentially activating them should be hidden from the client issuing the object request.

- *Location independent object references.* It must be possible to transparently reference objects, no matter where they are located. This must not interfere with potentially more complex architecture models. Less experienced developers must still be able to work on the level of an object-oriented programming model while experienced developers should be able to work on a software architecture model level.

- *Request handling.* In order to transparently hide evolution at the lowest abstraction level requests must be first order objects. This means that some kind of dynamic invocation interface [Sie96] is needed to construct and dispatch them. This very powerful mechanism, however, is bothersome for the application developer and very inefficient if the object on which an operation should be invoked is located in the same address space. For this reason a layer is needed that hides the inherent complexity, that makes it possible to optimize invocations, and that makes it possible to transparently map between compatible but slightly modified parameter lists in different versions of a capability (see also [Mae96]).

- *Exception handling*. Objects must be able to throw exceptions if some assumed precondition about their execution and operation context is not met, and if this violation prevents them from performing their operations properly.
- *Object life-cycle management.* Objects must be managed in several ways. It should be easy to create them, manage them during their different life-cycle phases, and finally delete them, either explicitly or based on some kind of garbage collection mechanism. In order to cope with client applications which leave references dangling, for example due to crashes, a possible strategy has to be more sophisticated than straightforward approaches like reference counting.
- *Version and variant management of interfaces and implementations.* Minimal evolution support requires both support for versioning object interfaces and implementations, as well as allowing several variants and versions to run in parallel. If systems grow larger it is also very important to be able to statically determine who uses which versions of which interfaces. This is important in order to find out which components are affected by a potential modification and which components have to be modified before support for an old version of a type can be discarded.
- *Object streaming.* It must be possible to flexibly convert "life" objects into passive data representations. In order to support this functionality a configurable capability is needed that makes it possible to generically access the state of an object. For example, a passive data representation of an object is required to transport it across networks or to display it in a debugger. Another potential client of such a capability is an algorithm that generically stores objects in a relational database.
- *Event notification.* Objects must provide means for clients to register their interest in being notified about changes of state. This state must be defined in the object's interface as its abstract state. Furthermore, interest registration mechanisms have to be provided. Event notifications can also be used to develop tools such as debuggers and monitors. For this application it must be possible to register interest in more generic or restrictive events such as the dispatching of a certain request to any object or the dispatching of a request with certain parameter values.
- *Transaction handling.* It must be possible to define larger units of execution, namely transactions, which are guaranteed to leave the system in a stable state, either doing nothing or finishing successfully. These transactions should work transparently on the object system in a similar way as they do with today's databases. The challenge of implementing transaction support for distributed object-systems is considerably bigger than for distributed databases because not all objects modified during a transaction must be located in a database. For this reason transaction mechanisms providing different degrees of reliability for different kinds of objects are needed.
- *Replication and group communication.* It should be possible to replicate services in order to achieve increased reliability and performance. To support replication, a reliable group communication protocol and messaging mechanism must be available.
- *Concurrency control.* Objects issuing a request should be able to specify whether a request is to be carried out synchronously or asynchronously. Furthermore, fine-grained synchronization, thread and process control should be possible.

- *Persistence.* Conceptually it has to be possible to make every object persistent in any kind of database. The persistence mechanism must, therefore, be designed to make it possible to store objects in any kind of media be it an object database, a relational database or a file.

- *Security.* It must be possible to control in a fine-grained way which objects owned by which user should be allowed to issue which request in order to access or manipulate a resource. It is impossible to design a generic security mechanism that works in every environment for every security enforcement strategy. For this reason a security mechanism has to provide a capability that makes it possible to implement any kind of security policy based on any kind of newly designed or available security mechanisms.

This list of requirements represents what has been discussed and (partially) specified in our architecture specification document. Many other possible services are not included, such as querying and licensing services as specified for CORBA 2.0 [OMG96].

Further functionality that cannot be represented by a single capability or service such as configuration management, explicit domain models and explicit reflective software architecture models will be elaborated below.

2.2 Support for large scale software development

Every project and every application of a certain size requires comprehensive tools to be understood, properly managed, developed and evolved. Not only does this include tools for editing, browsing, building, configuration management and distributed cooperative software development [Bis92, Bis94, Bis95], but also a set of comprehensive analysis and debugging tools. Debugging tools are particularly pertinent for the ever increasing complexity of distributed systems [Brü93, Deb88].

A software architecture specification and a conforming implementation must therefore explicitly provide hooks to allow smooth integration of tools. It should be possible to utilize these hooks to develop generic and project specific tools and to integrate them into the overall development process.

2.3 Support for evolution

The centralized information systems of large international companies such as banks are slowly turning into distributed systems either by expansion or by integration. They tend to be big and to become bigger over time. Complex dependencies between object implementations, services, middleware, legacy applications and systems emerge, which prohibit the simple replacing of single components. We have identified several requirements for system evolution that have to be addressed if we don't want future distributed applications to become legacy systems on the day of deployment.

- It must be possible to provide in parallel several versions of an object, component or service interface and implementation, and to have criteria at hand to decide which version to use for a given client.

- It must be possible to determine statically who is using which versions of a capability or a service.

- It must be possible to make new interfaces and implementations available at a prespecified point in time at all sites of a distributed application.

- It must be possible to specify dependencies between several interfaces and implementations in order to describe and configure domain models as higher level entities of evolution.
- It must be possible to test new implementations in the context of existing infrastructure and applications since it is most of the times hard, if not impossible, to build an adequate testing environment.

2.4 Abstractions for software architecture modeling

Large systems require adequate concepts, models, notations and techniques to be efficiently discussed, represented and operationalized. Objects and types are very powerful concepts, but they need to be extended and made more concrete in order to describe domains better. On a system domain level, concepts like pipe and filter better capture the intent of a design and its implementation than just "plain objects." The new field of software architecture deals with the definition of such concepts and languages to express them adequately [Sha96].

For example, some systems must provide a pre-defined level of responsiveness or throughput, must perform certain tasks within a given time limit, etc. Many of these guarantees can be formalized sufficiently in order to serve as evaluation and feedback criteria at runtime. To make these criteria subject to management, they have to be made explicit in a homogenous way. This might be done in several ways, for example by making the criteria accessible as service interfaces. However, representing concepts as interfaces is not enough: Such concepts needs a precise definition, and frequently additional models and techniques for using them in discussions and calculations. If adequate interfaces, concepts and techniques exist, it becomes possible, for example, to generically implement tuning based on support for load balancing and object migration, or to give users adequate feedback about expected system behavior in terms of performance, etc.

Sometimes, such new concepts must be introduced on a meta-level: It can be expected that the notion or Role will be an integral part of next-generation object models, so that we must prepare for its homogenous integration with existing object models today [Rie96c, Ree96, Mae96]. Introducing such new abstractions requires existing architecture support in order not to build everything anew.

2.5 Reusability of generic implementations

In order to make it possible to develop large-scale distributed object systems it is not sufficient to simply provide developers with an object broker and a set of predefined capability interfaces and services. It is as important to provide them with implementation level frameworks that generically implement those capabilities which are provided by all objects as well as capabilities provided by many domain objects. Furthermore, these frameworks must be readily configurable to allow the developers to explicitly state a set of policies for overall management of the objects involved.

The domain of distributed objects is generic to such a degree that large scale reuse is possible, but generally only within one management domain, for example within a single company. This is, because the implementation of many capabilities depends heavily on the available middleware components and their configuration which is often company specific and has grown over time.

2.6 Integration with existing infrastructure

New systems must integrate with existing legacy applications, middleware and operating systems. A new software architecture must both utilize existing functionality, because not everything can be invented from scratch, and integrate and run in parallel with existing systems, because those cannot be replaced at once.

2.7 Conclusions

The requirements discussed in this section are what we consider mandatory for an architecture supporting the development of any kind of global business objects. We know that it is hard to come up with designs and implementations that fulfill all requirements. Fortunately, this is not necessary from the beginning because most applications have only a subset of the requirements. If the main abstractions of architecture prove to be mature enough it will be possible to evolve the designs and implementation once it is necessary.

The next section presents the software architecture we designed to fulfill these requirements. It shows a single unifying theme, the use of reflection, which we use as a means for fulfilling the requirements. After the architecture descriptions we review how we think to meet the requirements.

3 Overview of the GBO/GNORF Architecture

GBO (Global Business Objects) is the name of the project carried out to prove the viability of the architecture for practical application in bank projects. GNORF (Global Networked Objects based on a Reflective Framework) is the preliminary name of the research project pursued at Ubilab. GNORF deals with the research aspects discussed above.

In this and the next two sections we review our architecture specification. This section gives an overview of how the different parts of the architecture fit together as depicted in Figure 1. The second section discusses the components of our object model and the third section presents the architecture of the virtual machine and its system architecture which is used to implement the object model.

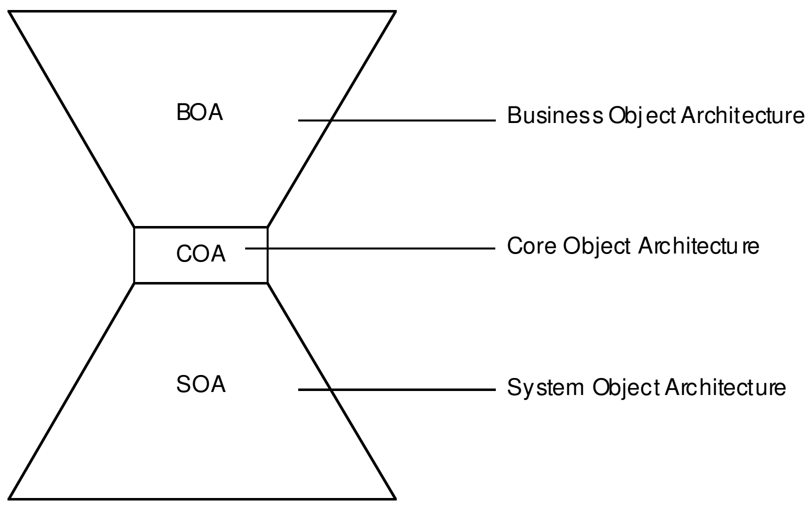

Figure 1: High-level view of the GBO/GNORF architecture

The basic idea underlying the GBO/GNORF Architecture is to develop a virtual machine for distributed object systems. It supports a meta-level architecture similar to CLOS [Kic93a, Kic93b] on top of which any kind of domain object model can be implemented. Objects are instances of types, and types control their instances by managing them and their implementations.

The architecture specification is based on type specifications which are independent of any particular programming language. In fact, the architecture specification defines its own object model using an IDL (interface definition language) which is imposed then on a concrete system environment and programming language object model.

Important features of this virtual machine are:

- It abstracts distribution by providing location independent object references.
- It provides the type AnyObject which serves as the supertype of all other types. It defines and generically implements many of the capabilities asked for in Section 2.
- Types are implemented as objects themselves. They handle the dispatching of operation invocations on their instances. This can be used to coherently integrate any kind of legacy implementation into a GBO/GNORF architecture implementation.

The hourglass of Figure 1 represents the main three layers of a GBO/GNORF based system.

- The topmost layer represents the Business Object Architecture (BOA) domain. Each business defines its own BOA. Every BOA consists of a number of types defining the supported business objects, their functionality and their relationships with each other.
- The middle layer, the Core Object Architecture (COA), provides the most important types of the virtual machine which define, for example, the generic capabilities and the meta-level architecture.
- The bottom layer, the System Object Architecture (SOA) implements the virtual machine types defined in the COA and offers further service interfaces which encapsulate existing computing environments.

Systems compliant with the architecture specification can be viewed as sets of distributed objects which interact through well defined interfaces. Each object represents an instance of a business or system domain concept and exerts proper behavior. Business or system domain concepts are represented and modeled as types the instances of which are the just mentioned business or system objects. This kind of architecture has many advantages. The most important of them are:

- It makes it possible to design and implement distributed business objects as sets of instances of types which utilize a virtual machine that takes care of most of the difficult aspects of distribution.
- Applications are portable between SOAs as long as all business objects are implemented based on COA types and services only, and as long as the different SOAs have similar quality-of-service profiles.
- The COA type AnyObject guarantees that all objects provide many capabilities such as persistence, migration, transaction support and event handling for which default implementations are inherited automatically.

- Applications are developed in a homogeneous world while most of the complexity of distribution is dealt with in the virtual machine. This diminishes the gap between analysis and design considerably.
- Instead of writing new code implementing the functionality of a type it is possible to bind it to legacy code. This means that the abstractions of legacy applications can be explicitly modeled as BOAs and exported into a homogeneous distributed object world.

We believe that requirements like support for large scale software development, evolution, and guaranteeing runtime behavior can only sensibly be achieved with a software architecture which is reflective in all its key abstractions. If this is not the case, one will always face situations which operate on implicit information and thus are based on potentially costly work-arounds.

4 GBO/GNORF Core Object Architecture (COA)

The COA defines the types and the semantics of the functionality the SOA provides to the BOA as explained above. This section starts with an overview of the COA types, their functionality and their relationships. This overview is followed by a number of subsections describing the key types in further detail.

4.1 Overview

All types inherit from *AnyObject*, which defines the capabilities offered by any object used in the context of the virtual machine. These are, among others: streaming, persistence, migration and event handling. Figure 2 shows the most important types which are part of the COA.

At the top of Figure 2, type AnyObject is displayed. Any other type must be a subtype of AnyObject, for example system level types like Request, Location or AnyWriter, or domain types like Customer, Portfolio or Account.

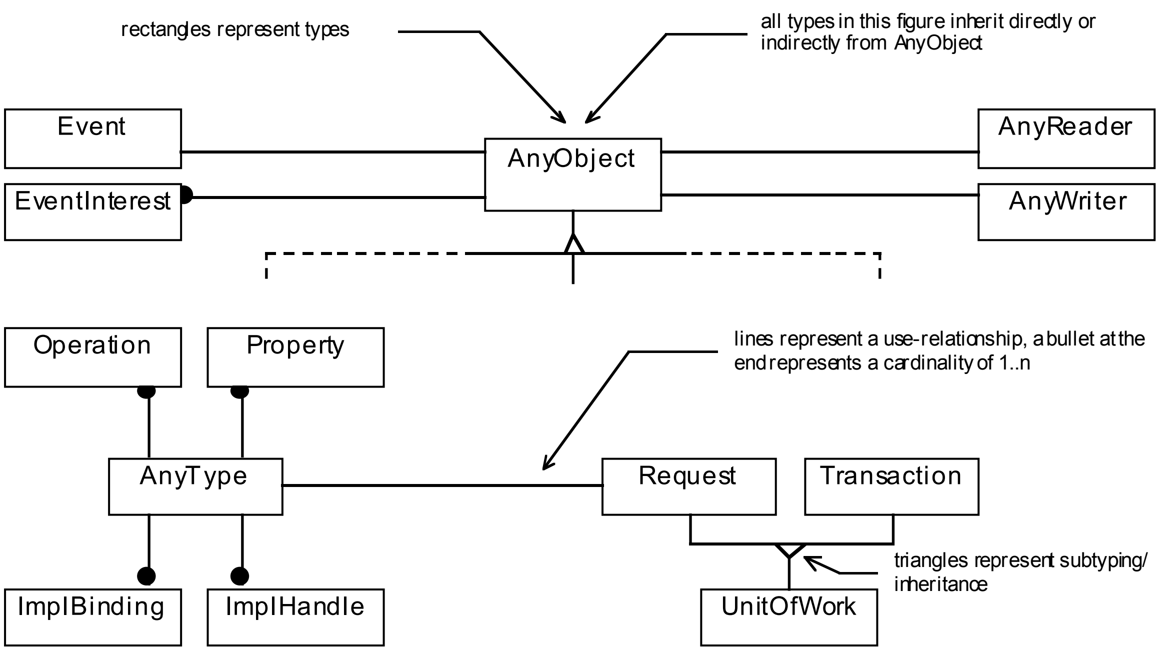

Figure 2: OMT diagram visualizing the central COA types and their relationships

Every type in the system is represented as an object. The interface of every type object is defined by *AnyType,* a subtype of AnyObject. Thus, type objects are instances of the AnyType type object. The AnyType type object is an instance of itself. Instances of type AnyType use instances of the types *Operation*, *Property*, *ImplBinding* and *ImplHandle* to explicitly represent interfaces and to bridge the gap between interfaces and the corresponding implementations.

Type objects manage their instances. Type instances are usually implemented using constructs native to the host language. In the case of an object-oriented programming language, the main implementation construct for implementing type instances are classes. A type object knows how its instances are implemented through specific classes.

A type object, for example type Customer, manages the execution of requests on its instances. A client provides both an object *Reference* and a *Request* object to the type object. The Request object defines the operation to be called and the Reference object defines the object on which the operation is to be executed. The type object dispatches the request to a proper implementation of the referenced object.

Each object can provide a passive data representation of itself, which is used for transporting objects between processes, for remote method invocation, for streaming objects to files, etc. Explicit *AnyReader* and *AnyWriter* objects decouple the object to be read or written from the reading and writing process and the concrete backend involved.

Every type defines in its interface which kind of events it announces and offers operations for clients to register interest in these events. An event results in an event notification of all interested objects. Event notifications can be dispatched either synchronously or asynchronously. An example of an asynchronous event is the notification that a certain operation has been carried out. A typical example of a synchronous event is the notification that a certain operation is going to be carried out. In the latter case, the notified object gets the chance to stop the invocation, for example to prevent a security violation, or to add a patch to the operation.

The types *Request* and *Transaction* are subtypes of *UnitOfWork* which defines their execution interface. UnitOfWork instances can have subunits and define a protocol for executing atomic actions. Further subtypes elaborate the interface for different actions, be it single requests or transactions based on two phase commit. The simple transaction protocol (commit and rollback) as well as the two phase protocol are used as mixin types so that there are transactional requests and two phase requests which support the respective behavior.

The specification for the COA addresses many more issues, including the meta-level architecture, persistence, security, etc. In the next subsections we focus on the key aspects of this architecture only and ask the reader to refer to [Bis96] for more details.

4.2 Types and implementations

An object's type is represented as a first class object which exists at runtime. It is responsible for most of what a compiler generates as code in statically typed languages: It instantiates, manages and deletes its instances. It checks, controls and dispatches calls to its instances, so that in principle every operation call is always routed via the type level. These type objects are, of course, instances of further type objects, so that a structurally reflective meta-level architecture results, much like the one found in CLOS [Kic93a].

Types can multiply inherit from further types, their supertypes, thereby becoming their subtypes. We strictly adhere to a behavioral subtyping discipline [Lis88, Lis93] with the only relaxation of allowing covariant redefinition [Mey92] the problems of which we intend to address through the help of explicit system and domain model type checking.

A type can manage several instance implementations which it chooses according to internal policies or based on external specifications, compare [Rie95a]. Implementations can be anything imaginable, including legacy applications—it is left to the type to dispatch an incoming request to an instance, and it can do so in what ever fashion it prefers. We have designed, though, a standard implementation handling and binding mechanism for the most common cases.

There is no need to define an equivalent of the Java interface concept since a strict interface and implementation hierarchy distinction [Kil92, Por93] is inherent in our type system.

4.3 References and requests

Objects know of each other via references. References are, from a client's point of view, black-box objects that serve as handles to the used object. A Reference object is interpreted by the type object of the referenced objects dynamic type. The type object manages the mapping between references and implementation objects. Thus, the type object has complete control over what it stores in it's reference objects.

It is transparent to a client where a referenced object is located. This information is managed by its type object or some appropriate manager object which is determined by the type object. The type object, therefore, defines both how its objects are distributed and what kind of information is stored in its references. This makes it easy to optimize distribution related performance aspects by parameterizing type objects with different distribution strategies.

The execution of an operation of an object is triggered by first creating an explicit Request object which contains the operation's name and its parameters, and then by telling the target object's type to execute the request on the instance indicated by a Reference object. This rather elaborate and inconvenient process is usually hidden behind convenience interfaces of the Reference object so that a developer invokes a regular method in the implementation. Behind the convenience interfaces, any proper optimization can be done, depending on the given circumstances. It is, for example, possible to invoke methods of local objects directly, without the expensive creation of the implicit request object and the dispatching through the type object.

An interesting issue to be considered is the level of granularity on which the discussed object model is applied. Two extremes can be thought of. On the one hand, the model is applied to its fullest extent, with every concrete object being handled by the virtual machine. Then, every request to any object is dispatched by that object's type object, making the system very flexible. Thereby, the architecture's object model is fully adopted and the host language's object model is ignored.

In the other case, the virtual machine serves as a remote method invocation mechanism which bridges between heavyweight implementation subsystems. These subsystems, just like legacy applications, handle most of the arising issues internally using the programming language's object model. They rarely make use of the virtual machine but only use it for communication calls between the subsystems.

Our request execution approach is conceptually similar to CORBA's dynamic invocation interface [Sie96]. However, our intention is not only to provide a dynamic means for creating and executing requests, we also want to make request and reference objects explicit for further operationalization, for example as part of a transaction.

Requests can be synchronous or asynchronous, with the second option being hard to fully implement, since it requires a full multi-threading model. Exceptions or out parameters are propagated back with the executed request object. Some discussions of similar and related issues can be found in [Wol96, Arj96].

4.4 Objects and values

Value types like Integer, Float, and String are kept separate and are not derived from AnyObject. Conceptually, values are not objects, but the primitive elements of computation from which objects can be built. Values have no identity. Since they are always copied, they are always local, and no referential integrity has to be maintained. For value types only standard data representation formats have to be defined so that they can be transported between processes.

A distinction between *primitive* and *non-primitive* value types can be made. A primitive value type is a value type for which a direct language mapping exists, including integers, floating point numbers, characters, etc. Non-primitive value types are types like Currency, Amount, SocialSecurityNumber, etc. Being non-primitive means that values have to be implemented as objects, because they have no direct language mapping. This allows the introduction of domain specific value types.

Objects on the other hand have an identity. They define a state space, with each dimension representing a property and being identified by a property name. Every name can be bound to a value, including references to further objects. An object never changes a value but rather replaces the value bound to a property name with another value.

4.5 Further important capabilities

Types model their interface explicitly as operation and property objects. Operations and properties model the abstract state space of the type's instances which have to be met by every type instance implementation. This explicit and canonical model lets client objects register interest in particular changes of an object's state. A proper event notification mechanism takes care that they are informed about changes in that particular aspect of the changed object, compare [Gar93, Rie96a]. Requests and transactions are subtypes of UnitOfWork, which, together with transaction protocol and two phase protocol classes, defines the interface to different request and transaction variants. Further types are provided for object streaming, persistence, security, etc. but are not discussed here. More detailed descriptions can be found in [Bis96].

5 GBO/GNORF System Object Architecture (SOA)

A detailed specification of a concrete SOA is a document with well above 100 pages. The goal of this section is to give a rough overview of how a SOA implementation refines the hourglass architecture presented at the beginning of Section 3 and to give an overview of the activities in the area of SOA development.

5.1 System architecture and infrastructure mapping

In the SOA, the hourglass architecture of Figure 1 is refined to a layered architecture [Bus96]. Business domain type implementations build on standardized domain types and virtual machine types (the COA types of Section 4). The virtual machine type implementations build on SOA specific services, which in turn build on a specific computing environment. Note, that an implementation always refers to types which it is based on and usually should not make use of a specific implementation.

A COA-compliant SOA implementation depends on several dimensions: It depends on the chosen language, the available frameworks and libraries, and the existing computing environment. This triple defines the implementation space. For every language in every environment a new SOA implementation has to be carried out. To avoid this proliferation of variants one should confine oneself to a set of standard languages and middleware products.

Alternatively, one can design rather light-weight clients which are restricted to dispatching request objects to referenced objects in heavy-weight servers which incorporate the domain model and a fully implemented SOA. The only functionality that has to be implemented for such a light-weight SOA is a Reference type and the mechanics of forwarding Requests to the server. This approach seems particularly feasible with, for example, Java applets for the world-wide web. Only a very small SOA implementation will have to be transferred to the WWW-browser.

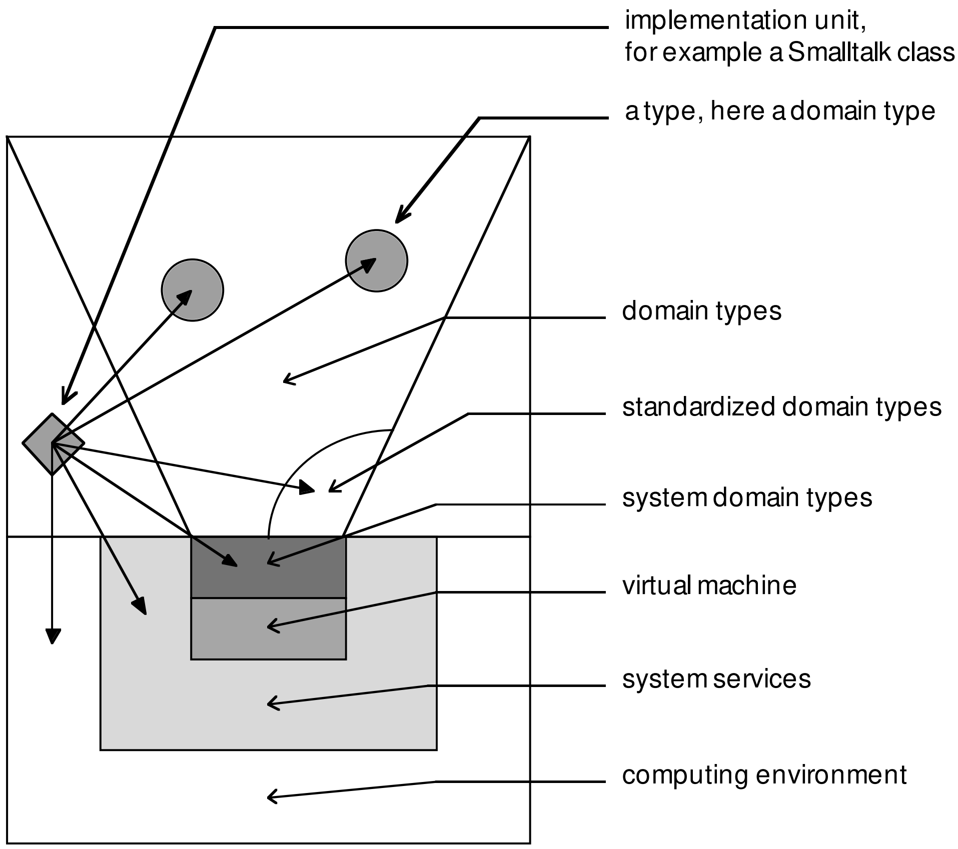

Figure 3: System-level structure and layering

We will now examine the layers depicted in Figure 3 in more detail, starting from the bottom:

The bottom layer represents the existing computing infrastructure. This can be applications, or existing middleware like network software and transaction monitors, or databases like Oracle or ObjectStore, or CORBA compliant object request brokers like Orbix or ObjectBroker. This is the “computing environment” layer as depicted in Figure 3.

On top of the computing environment, system services required for the implementation of the virtual machine and domain applications are developed. These system services represent a convenient and, in terms of abstraction, adequate encapsulation of the underlying middleware in order to make the virtual machine more reusable across platforms and to make it more robust with respect to changes in the environment. For example, one might use CORBA services, or even traditional databases, networking services, etc.

The system domain types represent the interfaces to the virtual machine which implements them based on the underlying system services. These types are defined in the COA, the core object architecture. These types and the layers below, that is the virtual machine implementation itself, the system services, as well as the integration with the existing computing environment represents the SOA, the system object architecture.

Above the COA and SOA, the different Business Object Architectures (BOAs) are defined. A BOA always depends on the business domain, and nothing can be said about its requirements other than the system-level requirements already stated in Section 2.

While it seems best to let implementations refer to types only and let a BOA type implementation never refer to anything else than a COA type, it cannot be hold upright in practice. Therefore, we allow developers to implement BOA types based on system services and even the computing environment while at the same time making the dangers resulting from such an implementation explicit. Migrating objects of types which require implementations bound to a specific computing environment is not transparently possible and therefore limits the power of the architecture implementation and the resulting system. However, flexibility, completeness and economics have always to be weighted and measured against each other.

5.2 Development activities

There are currently two different SOA development activities. In the GBO project a prototype has been implemented in Smalltalk which shows that it is possible to implement a basic SOA in Smalltalk in about two human months. In its current version, the prototype is only partially conformant to the actual architecture specification and omits some of the harder implementation problems.

In the GNORF project we just started to implement a SOA prototype in Java. In implementing this prototype we pursue two goals. One of them is to develop a working test environment to experiment with various evolution support strategies. The other goal is to develop an extremely light-weight Java SOA that can be run in various light-weight Java machines such as a WWW browser or a dedicated Internet terminal. Based on this, it should be possible to quickly develop user interfaces that have all advantages of the WWW and Java.

Along the way, we expect to evaluate further research issues, for example how Java code shipping can be utilized for providing location transparent implementations, and how well the touted internet architecture, WWW-browsers and Java applets, work for large scale systems.

6 Fulfilling the Requirements

In this section we review how we think that the architecture specified in Section 3-5 fulfills our requirements discussed in Section 2. Some of these requirements were obviously easy to satisfy while others will constitute the major part of our research work.

- *Set of basic capabilities and services.* Our architecture specification is built on top of a reflective type system which defines many useful capabilities. These capabilities are provided by types at a well-defined point in the type hierarchy and therefore guarantee that subtypes will offer these capabilities. Some of these capabilities are considered to be so important that they are guaranteed to be provided by any object and therefore are part of the interface of AnyObject. Next to capabilities, our architecture specification names a number of services which are needed for a large distributed system [Bis96].

  The definition of these capabilities and services was based on Genesis' and Ubilab's experience, for example with object-oriented frameworks like ET++ [Wei94], Smalltalk [Gol89, Smi95], Sniff and Beyond-Sniff [Bis92, Bis94, Bis95], and the Tools and Materials Frameworks [Rie95b, Rie96b, Bäu96]. We can draw here on CORBA as well. In this respect, it was a straightforward work, even though experience with the on-going projects will surely lead us to refine, extend and change the actual capabilities and services specification.

- *Support for large scale software development.* The reflective architecture allows for very fine-grained control and manipulation of the various aspects of a distributed system. We consider this as a key which provides the hooks for general as well as project specific tools to be incorporated into the overall software development process.

- *Support for graceful evolution.* Again, the reflectiveness of the type system allows us to analyze and manipulate every important aspect of the type system, including a type's interface and implementation binding. This extremely flexible mechanism allows us to replace type interfaces and implementations both statically and dynamically and therefore provides us with the basic building blocks to support system evolution.

  It is not sufficient, however, to evolve and replace single types. It must be possible as well to evolve sets of collaborating types, that is domain models. To achieve this, it must be possible to both specify such a domain model as a modeling entity of its own as well as to evolve it as such an entity. We will therefore develop proper domain modeling notations and tools as well as transactions on the type and domain model space. Transactions of this kind sound complicated at first but since all relevant abstractions are represented as objects, it should be possible to realize it as regular transactions.

- *Introducing new abstractions of software architecture.* This requirement was motivated by the need to make domain specific abstractions of software architecture explicit in order to control and manage them, for example for guaranteeing a certain throughput or performance, maximum time of execution, or for optimizing runtime behavior, for example through load balancing.

  Moreover, an explicit software architecture model lets us view, discuss and check the system at a higher level than basic object-oriented design allows us to do. It is clearly understood today that object-orientation itself does not deliver adequate abstractions for modeling large distributed systems. Further concepts, including architectural styles and patterns [Gar94, Sha95, Gam95, Cop95, Vli96] are needed to develop an adequate design vocabulary and make it explicit in software.

- *Reusability of generic implementations.* The clear separation of implementation from types and their clear placement in the type hierarchy allowed us to provide reusable implementations already for very generic types and therefore helps to fulfill our requirement of implementation reuse and therefore increased productivity.

- *Integration with existing infrastructure.* Again, the clear separation of implementations from their types makes it possible to hide any existing infrastructure behind it. The SOA defines several levels at which existing legacy applications and middleware can be wrapped. The SOA works as a broker, encapsulating existing infrastructure and making it available to a large number of clients which should not have to know anything about the underlying implementation.

A final implicit requirement was not to make developers change their thinking model. They still should be able to work on an object-oriented modeling and programming level. The architecture specification allows them to do so as long as it is sensible and possible. In those aspects in which object-orientation is not a sufficient means of describing the system, developers will have to switch to proper software architecture models and their specialization in their respective application context.

7 Related Work

The GBO/GNORF project draws on different areas of computer science, namely on software architecture, object-oriented systems, frameworks and distributed systems. For this reason it is impossible to do a more or less complete discussion of related work in this paper. Such a discussion would be a book about software architectures in general and distributed object systems in particular.

In this section we focus on the area of distributed objects and discuss both homogenous distributed object systems as well as CORBA, the OMG standard to integrate heterogeneous systems.

In homogenous distributed object systems most aspects of distribution are covered by the infrastructure and a developer implements distributed applications in almost the same way as he or she implements single process applications. Examples of homogenous distributed object systems are Portable Distributed Objects (PDO) from NeXT and Distributed Smalltalk from IBM. Both products provide a high degree of location independence for objects on a programming language level. From the point of view of simplicity these systems are almost optimal.

The inherent problem of homogenous distributed object systems is that distribution is handled by the underlying virtual machines or runtime systems. This makes it almost impossible to modify central design decisions such as what kind of information is stored in a reference and how distributed objects of a type are best managed and garbage collected. While this is no problem for small to medium sized applications it becomes a central issue in developing globally distributed large-scale applications.

In the GBO/GNORF the developer can decide on the optimal mix of ease of use and flexibility. The developers of new types (business objects) should be aware of the potential of the type system and should optimize distribution behavior of important types extensively. For the application developers that are applying the business objects it should be almost transparent that they are distributed.

In comparing CORBA with GBO/GNORF we can clearly recognize the two different approaches. CORBA has grown over many years from a remote method invocation mechanism into a standard for that tries to cover almost all aspects of distributed object systems. Today, there are not only reasonable standards but also implementations of it.

The inherent problem of CORBA is that something planned to be a simple remote method invocation mechanism has grown into a standard that tries to cover such diverse aspects as management infrastructure, basic services and capabilities, and application area specific frameworks. Due to the nature of standardization processes it was not possible (or intended) to revise earlier standardized aspects while working on extensions. This has resulted, for example, in a sometimes confusing terminology and in a number of capabilities and services that have grown into a jungle of interfaces instead of an elegant framework. In GBO/GNORF we had the advantage to start from scratch which gave us the possibility to come up with a homogenous and much easier comprehensible architecture.

We think that it is possible to provide considerably superior support for the development of distributed applications than what is provided by CORBA compliant systems. Nonetheless we appreciate the amount of work that has gone into the infrastructure of CORBA and consider it sensible to use CORBA as the lowest layer on which an infrastructure that is compliant with our architecture can be developed.

8 Conclusions and Future Work

In this paper, we have presented an overview of the GBO/GNORF project from a research perspective. We have identified, described and motivated some of the key requirements for a world-wide distributed object-oriented software architecture. These requirements include the definition of a set of guaranteed capabilities and services, support for large scale software development in terms of tools and hooks for tools, explicit modeling and representation of software architectures, support for evolution of single types as well as frameworks, reusable generic implementations, and ways of integrating an architecture specification conforming implementation with an existing infrastructure.

We have also presented how we intend to fulfill these requirements by means of a reflective distributed object-oriented architecture realized through a virtual machine and accompanying services. Such a reflective architecture contains all the “primitives” which we consider to be important to reach the goals.

As of today, prototypes have proven that the implementation of several key elements of the meta-level architecture is feasible. However, for many of the more advanced aspects of the architecture a reference implementation has still to be done. Such aspects are object synchronization, concurrency, replication and long transactions – each an interesting research topic on its own.

Our short to medium range focus will rest on supporting type evolution as one of the most pressing problems for systems of the envisioned size and required reliability. In parallel, we will work on domain model representations and their operationalization. Finally, we will work on introducing and operationalizing higher level software architecture abstractions than types and relationships in order to more adequately help modeling business domains.

Acknowledgments

We would like to thank Dirk Bäumer, Dieter Schlegel, André Weinand and Heinz Züllighoven for reading and commenting on the paper.

References

[Arj96] Arjomandi E, O'Farrell WG, Wilson GV: Smart Messages: An Object-Oriented Communication Mechanism for Parallel Systems. *COOTS-2, Conference Proceedings.* USENIX Association, 1996

[Bäu96] Bäumer D, Gryczan G, Lilienthal C, Riehle D, Strunk W, Wulf M, Züllighoven H: The Tools and Materials Approach—Analysis, Design and Construction of Interactive Object-Oriented Systems. *To be submitted for publication*

[Bis92] Bischofberger WR: Sniff – A Pragmatic Approach to a C++ Programming Environment. *USENIX 1992 C++ Conference Proceedings.* USENIX Association, 1992

[Bis94] Bischofberger WR, Kofler T, Schäffer B. Object-Oriented Programming Environments: Requirements and Approaches. in *Software–Concepts and Tools,* Vol. 15, No. 2, Springer Verlag, 1994

[Bis95] Bischofberger WR, Kofler T, Mätzel K, Schäffer, B: Computer Supported Cooperative Software Engineering with Beyond-Sniff. SEE '95, *Conference Proceedings* (7th Conference on Software Engineering Environments)

[Bis96] Bischofberger WR, Guttman M, Riehle D, Sturmer C: *Global Business Objects System Object Architecture.* Union Bank of Switzerland, 1996

[Brü93] Brügge B, Gottschalk T, Luo B: A Framework for Dynamic Program Analyzers. OOPSLA '93, *Conference Proceedings.* see also *ACM SIGPLAN Notices,* Vol. 28, No. 10, 1993

[Bus96] Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M: *Pattern-Oriented Software Architecture—A System of Patterns.* John Wiley and Sons Ltd, Baffins Lane, Chichester, 1996

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

[Deb88] Miller B, LeBlanc T (eds.): *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging. ACM SIGPLAN Notices,* Vol. 24, No. 1, 1989

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

[Gar93] Garlan D, Notkin D. Formalizing Design Spaces: Implicit Invocation Mechanisms. VDM '91, LNCS 551, *Conference Proceedings.* Berlin, Heidelberg, Springer-Verlag, 1991

[Gar94] Garlan D: What is Style? *Proceedings of the Dagstuhl Workshop on Software Architecture*, February 1995

[Gen95] Genesis Development Corporation. *Enterprise Object Architecture*, October 1995

[Gol89] Goldberg A, Robson D. *Smalltalk-80: The Language.* Reading, Massachusetts, Addison-Wesley, 1989

[Kic91] Kiczales G, des Rivières J. Bobrow DG: *The Art of the Metaobject Protocol.* Cambridge, Massachusetts, The MIT Press, 1991

[Kic93] Kiczales G, Ashley JM, Rodriguez Jr JM, Vahdat A, Bobrow DG: Metaobject Protocols: Why We Want Them and What Else They Can Do. *Object-Oriented Programming: The CLOS Perspective*. Cambridge, Massachusetts, MIT Press, 1993

[Kil92] Killian MF. Trellis: What we have learned from a strongly typed language. *OOPSLA '92, Conference Proceedings.* see also *ACM SIGPLAN Notices,* Vol. 27, No. 10, 1992

[Lew95] Lewis T: *Object-Oriented Application Frameworks.* Greenwich, CT, Manning, 1995

[Lis88] Liskov B: Data Abstraction and Hierarchy. OOPSLA '87 (Addendum), *ACM SIGPLAN Notices,* Vol. 23, No. 5, 1988

[Lis93] Liskov B, Wing J: A New Definition of the Subtype Relation. LNCS 707, *ECOOP '93, Conference Proceedings*. Berlin, Heidelberg, Springer-Verlag, 1993

[Mät96] Mätzel K, Bischofberger W: Evolution of Object Systems or How to Tackle the Slippage Problem. *This volume*

[Mey92] Meyer B: *Eiffel. The Language*. New York, London, Prentice-Hall, 1992

[OMG96] Object Management Group. *CORBA 2.0, Common Object Services and Common Facilities Specification.* 1996

[Por93] Porter III HH: Separating the Subtype Hierarchy from the Inheritance of Implementation. *Journal of Object-Oriented Programming,* Vol. 4, No. 9, 1992

[Ree96] Reenskaug T: *Working with Objects.* Manning, 1996

[Rie95a] Riehle D: How and Why to Encapsulate Class Trees. OOPSLA '95*, Conference Proceedings*

[Rie95b] Riehle D, Züllighoven H: A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. in [Cop95]

[Rie96a] Riehle D: The Event Notification Pattern—Integrating Implicit Invocation with Object-Orientation. *Theory and Practice of Object Systems,* Vol. 2, No. 1, 1996

[Rie96b] Riehle D, Schäffer B, Schnyder M: Design of a Smalltalk Framework for the Tools and Materials Metaphor. *Informatik/Informatique,* Vol. 3, 1996

[Rie96c] Riehle D: Describing and Composing Patterns Using Role Diagrams. *This volume*.

[Sha95] Shaw M: Comparing Architectural Design Styles. *IEEE Software,* Vol. 12, No. 6, 1995

[Sha96] Shaw M, Garlan, D. Software Architecture: Perspectives on an Emerging Discipline. Prentice-Hall, 1996

[Sie96] Siegel J (ed.): *CORBA: Fundamentals and Programming.* Wiley, 1996

[Smi95] Smith DN: *IBM Smalltalk: The Language.* Redwood City, California, Benjamin Cummings, 1995

[Vli96] Vlissides JM, Coplien JO, Kerth NL: *Pattern Languages of Program Design 2*. Reading, Massachusetts, Addison-Wesley, 1996

[Wei94] Weinand A, Gamma E: ET++ — a Portable, Homogenous Class Library and Application Framework. *Computer Science Research at Ubilab*. Konstanz: Universitätsverlag Konstanz, 1994

[Wol96] Wolrath A, Riggs R, Waldo J: A Distributed Object Model for the Java System. *COOTS-2, Conference Proceedings.* Usenix Association, 1996