Architecture Support for Global Business Objects: Requirements and Solutions

Walter Bischofberger[1], Michael Guttman[2] and Dirk Riehle[1]
[1] Ubilab, Union Bank of Switzerland, Bahnhofstrasse 45, 8021 Zürich, Switzerland
[2] Genesis Development Corporation, 10 North Church Street, 4th Floor West Chester, PA 19380 USA
bischofberger@ubilab.ubs.ch, mguttman@gendev.com, riehle@ubilab.ubs.ch

Abstract

The development of world-wide distributed object-oriented systems poses a considerable number of hard questions. In this paper, we summarize these questions as a set of requirements which we consider to be important for a software architecture to be successful, and we present our solution for such a software architecture. Our main conclusion is that such a software architecture must be reflective in all its key abstractions in order to allow analyzing and operationalizing its properties. A mainstream banking project is on its way which conforms to this architecture. At Ubilab, we are focusing on the research aspects of the project like enabling smooth evolution and explicitly modeling and operationalizing the software architecture at runtime.

1 Introduction and Motivation

The Union Bank of Switzerland (UBS) is a large globally operating bank the operations of which 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, UBS is developing a homogenous object-oriented software architecture for both wrapping and integrating legacy systems and providing a common base for new applications.

In this paper, we present the main requirements for such an architecture as well as our solutions. Key requirements for this architecture are that it must explicitly support evolution from the very first day, must flexibly utilize and integrate existing and new middleware, and must provide information to analyze the architecture and to operationalize it, for example to guarantee pre-specified runtime behavior. We approach this goal with a distributed object-oriented virtual machine based on a small number of reflective key abstractions. We use this virtual machine to provide flexible evolution support and an operationalized software architecture model that lets components analyze and control the system.

UBS is undertaking this effort in form of a regular banking project, for which Genesis Development Corporation is consulting. The authors of this paper are three of the four authors of the key software architecture specification document [1]. Earlier this year, a first prototype has shown the feasibility of the approach in a limited setting, and by the end of the year we should have a proof of its viability in a world-wide distributed context. Ubilab, the research laboratory of UBS, is focusing on the research aspects of the project.

2 Requirements

The development of world-wide distributed object systems poses a number of requirements. We identify the following main categories, which we detail in the following:

- set of basic capabilities and services;
- support for large scale software development;
- explicit modeling of software architecture;
- support for graceful evolution;
- integration with existing infrastructure.

2.1 Set of basic capabilities and services

Every project and every application must provide certain capabilities and requires certain services to build upon.

A *capability* offered by an object is some functionality which makes it usable for specific clients. Usually, an object offers more than one capability which makes it useful for clients in different contexts. A capability is expressed as an interface. Types the instances of which have to offer this capability inherit from this interface. Examples are the capability of an object to provide a passive data representation of itself or the capabilitiy to announce events about state changes.

A *service* provides some useful functionality to an unknown number of clients which cannot be predicted in advance. Usually, a service object focuses on providing a single service as opposed to providing a number of different capabilities. A service is expressed as an interface which clients directly use. Examples of services are naming and transaction management.

Some capabilities and services are domain dependent, some are not. They should be provided as homogeneously as possible. Our experience indicates that a set of basic capabilities should by offered by any object in the system. Moreover, there are many mandatory services for any large system, especially in the banking domain. We do not detail their description, but leave this to more elaborate papers [2].

2.2 Support for large scale software development

Every project and every application of a certain size requires comprehensive tools to be properly managed, developed and evolved. Not only does this include tools for editing, browsing, building, configuration management and distributed cooperative software development, but also a set of comprehensive analysis and debugging tools. Debugging tools are particularly pertinent for the ever increasing complexity of distributed systems [3], [4].

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

2.3 Support for evolution

Systems in the banking and other domains are distributed systems which tend to be big and to become bigger over time. Complex dependencies between object implementations, services, middleware, legacy applications and systems emerge, which prohibit simple replacement of single components. At the same point in time, several object and component versions might be required to be available. In addition, it must be possible to test new extensions in the existing infrastructure. Thus, evolution must be considered from the very first day.

2.4 Guaranteed runtime behavior

Often, some pre-specified runtime behavior has to be guaranteed. Systems must provide a certain level of responsiveness or throughput, must perform certain tasks within a given time limit, etc. Many of these guarantees can be sufficiently formalized to serve as evaluation and feedback criteria at runtime so that manager components can take care of them. The information needed for such tasks must be made explicitly available so that it can be analyzed and so that further operations can be reliably based upon this analysis.

2.5 Integration with existing infrastructure

It is well-known that 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.

2.6 Conclusion on requirements

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 workarounds. We have therefore designed a reflective software architecture which we present in the next chapter.

3 The GBO/GNORF Approach

We now review our architecture specification and discuss how we think it fulfills the requirements. GBO (Global Business Objects) is the banking project name, and GNORF (Globally Networking Objects based on a Reflective Framework) is our preliminary name for the research project. GBO deals with implementing the basic architecture for generic application support, the first one of which is a prototype for managing high risk portfolios. GNORF deals with the research issues, that is flexible evolution support and explicit modeling and operationalizing the software architecture model at runtime.

3.1 Object model and software architecture

The key abstractions of our software architecture are Type, Implementation, Request and Reference, followed by a larger set of further types (depicted in figure 1). The software architecture makes these types first class abstractions, that is in a running system they are represented as objects. They form the heart of a virtual machine on which more elaborate services and frameworks are built. The implementations of the key types may be built on top of existing services which in turn may rest on existing middleware. For example, the migration of Request objects might reuse a general object migration service which in turn might use a CORBA compliant object request broker.

The basic architecture can be described as a single rooted type hierarchy. In this view, the root type is AnyObject, which

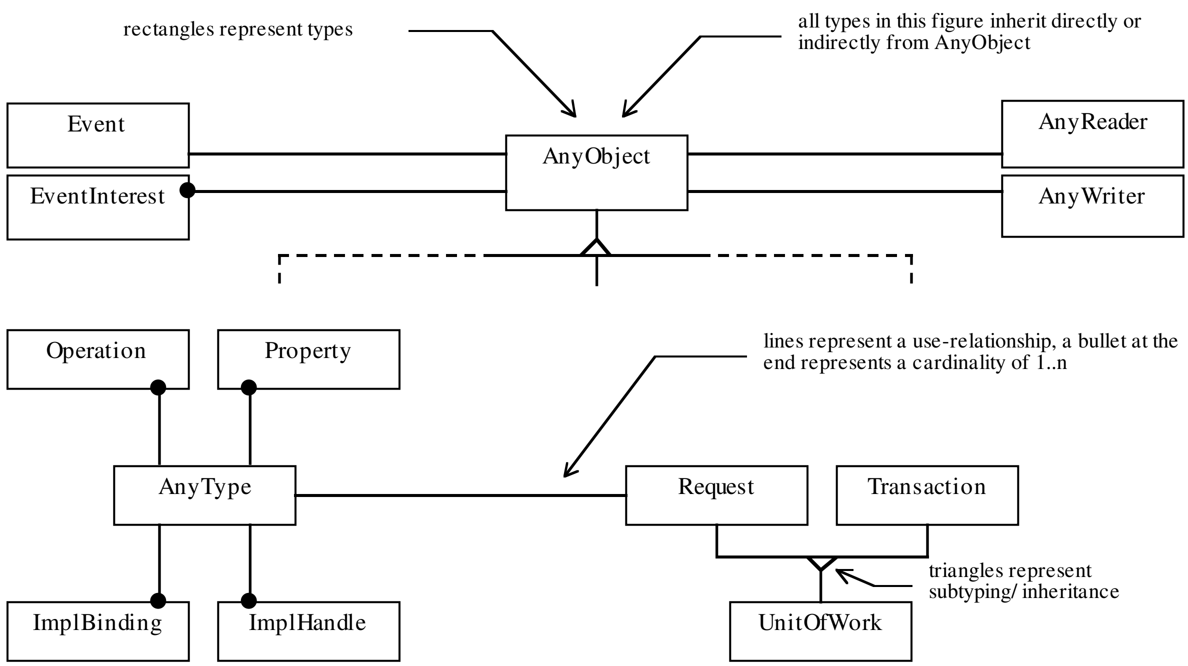

Figure 1: OMT diagram visualizing some central types and their relationships

defines the basic capabilities provided by any object. Instances of AnyObject can be referenced for purposes of remote operation calls, migrated to places where proper implementations exist, monitored using eventing, etc. Each type is represented as an object, that is an instance of type AnyType, which is itself a subtype of AnyObject. AnyType instances provide operations for plugging in different implementations and for fine-grained manipulation of type interfaces together with their versioning. Essentially, this is similar to a structurally reflective metalevel architecture like the one of CLOS [7]. It is organized as a traditional framework but does not face the problems programming language based frameworks do.

This reflective meta-level is used not only to explicitly represent single domain and business concepts as types, but also to represent explicit domain and business models as sets of types interacting according to the domain and business semantics. The same things that can be done with single type instances should also be possible to be done with more complex model instances. For example, it should be possible to migrate object graphs as a whole or to type check them for conformance with a model. The virtual machine is also part of a domain, namely the system domain, so that this applies to all levels.

Value types like Integer, Float, and String are kept separate and are not derived from AnyObject. This also includes non-primitive value types like Date and SecurityTicket. Values are not objects, since they are always copied, that is they are always local, and no referential integrity has to be maintained. Essentially, for value types only standard data representation formats have to be defined so that they can be transported between processes.

Type objects handle the references to their instances, define their interpretation and manage their name space. Furthermore, they carry out the dispatch required when clients issue requests to their instances. Both references and requests are represented as objects, with Request being a subtype of AnyObject, and Reference being a value type. Explicit and reflective type objects are our first key to managing type interface and implementation evolution.

Transactions are based on explicit Request objects. Transactions provide an interpretation context for execution, and are either guaranteed to fully succeed or fail without leaving garbage. This can be used for system evolution: Installing new versions of a set of types and type implementations is treated as a transaction which either completely succeeds or fails, always leaving the system in a stable state. Combined with code shipping this is our second key to system evolution.

3.2 Fulfilling the requirements

How does the proposed architecture fulfill the requirements posed in section 2?

- By clearly defining a type hierarchy with explicit protocols, it is clear at which level which capabilities are guaranteed to be available. They can be implemented based on further services.
- The reflective architecture allows analyzing and operationalizing every relevant abstraction and therefore provides support for almost all tools imaginable.
- The reflective architecture provides the "primitives" for reflective higher level models of software architecture. Without analyzable and operationalized type and request objects, it does not make sense to talk about analyzable and operationalizable software architecture models.
- The reflective architecture provides the hooks for fine-grained manipulation of type interfaces and implementations and therefore is a fine starting point for evolution.

Reflection is used here as a means for fulfilling the requirements posed in section 2. It is not an end in itself, but rather the best solution we know of that provides us with the required flexibility to develop support for type and domain model evolution and to explicitly model a system's software architecture.

4 Related Work

Compared to CORBA [8], [11], our approach is much more flexible at the initial expense of static typing. CORBA provides a well elaborated object model and a set of well engineered service specifications, but it lacks concepts like explicit and operationalized type and request objects which we consider to be key to our problems and their solutions. SOM seems to be more similar to our approach, so that we are investigating whether we can instrument it for our purposes.

The introduced reflectivity allows us to tackle evolution problems much easier than CORBA does. Our type objects allow detailed control over interpretation of type specific references and optimize life-cycle issues (no factories). The interpretation of requests as objects and their handling through reflexive types lets us design and implement transactions in an eased manner and apply them to every object in the system, including the type system and domain models.

5 Conclusions

We have presented a reflective object model which can be implemented as a virtual machine running on several different platforms, and have shown how we think it will help us fulfill a number of difficult requirements for large distributed software systems. The reflectivity of the model makes it possible to introduce high-level abstractions of software architecture at runtime. We expect them to be better suited to tackle some of the harder problems of such systems than those of the basic object paradigm. In particular, they will help us to address evolution issues and guaranteeing runtime behavior.

The high-level abstractions we will be using are still evolving: We are working on graph-based specifications for object-oriented (business or system) domain models to support evolution on both a fine-grained and a coarse-grained scale. We are considering using the component and connector model of Shaw et al. [10] as the basic metamodel for software architecture operationalization.

It is worth noting that the presented architecture heavily draws on a wide range of architectural styles like implicit invocation, pipes and filters, data abstraction and layering [6], [10]. All are based on object-orientation as the dominating modeling and implementation paradigm. For operationalized architecture models, we will have to draw on all of them.

Currently, a first prototype has shown that the metalevel architecture can be implemented efficiently, but for many of the more advanced aspects of the architecture, a reference implementation must still be done. In particular, this includes aspects like object synchronization, concurrency, replication and long transactions—each an interesting research topic of its own.

Our short to medium range research focus will rest on supporting type evolution as one of the most pressing problem for systems of this 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 those of the basic object-oriented system model.

References

[1] Walter Bischofberger, Michael Guttman, Dirk Riehle and Carl Sturmer. *Global Business Objects System Object Architecture.* Zürich, Union Bank of Switzerland: 1996.

[2] Walter Bischofberger, Michael Guttman and Dirk Riehle. "Global Business Objects." In preparation, 1996. Contact the authors for a copy.

[3] Bernd Brügge, Tim Gottschalk and Bin Luo. "A Framework for Dynamic Program Analyzers." OOPSLA '93, *Conference Proceedings.* See also: *ACM SIGPLAN Notices* 28, 10 (October 1993): 64-82.

[4] Barton Miller and Thomas LeBlanc (Editors). *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging.* ACM SIGPLAN Notices 24, 1 (January 1989).

[5] David Garlan (Editor). *Proceedings of the First International Workshop on Architectures for Software Systems.* CMU Technical Report, CMU-CS-95-151, 1995.

[6] David Garlan, Andrew Kompanek, Ralph Melton and Robert Monroe. "Architectural Style: An Object-Oriented Approach." Submitted for publication, 1996.

[7] Gregor Kiczales, J. Michael Ashley, Luis H. Rodriguez Jr., Amin Vahdat and Daniel G. Bobrow. "Metaobject Protocols: Why We Want Them and What Else They Can Do." *Object-Oriented Programming: The CLOS Perspective.* Edited by Andreas Paepcke. Cambridge, Massachusetts: MIT Press. 101-118.

[8] Robert Orfali, Dan Harkey and Jeri Edwards. *The Essential Distributed Objects Survival Guide.* Wiley, 1995.

[9] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young and Gregory Zelesnik. "Abstractions for Software Architecture and Tools to Support Them." *IEEE Transactions on Software Engineering* 21, 4 (April 1995): 314-335.

[10] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[11] Jon Siegel (Editor). *CORBA-Fundamentals and Programming.* Wiley, 1996.

[12] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead Jr., Jason E. Robbins, Kari A. Nies, Peyman Oreizy, and Deborah L. Dubrow. "A Component- and Message-Based Architectural Style for GUI Software." *IEEE Transactions on Software Engineering* 22, 6 (June 1996): 390-406.