An Investigation into the Mechanisms that allow CORBA to preserve Strong Typing

by

David Lievens

A dissertation submitted in partial fulfilment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Glasgow

GLASGOW

December 2000

ProQuest Number: 13818883

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 13818883

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

// Abstract

The Common Object Request Broker Architecture (CORBA) is a middleware specification. It aims at transparently extending programming languages to enable access to objects that are situated in different address spaces. Extending strongly typed languages raises the question whether the extension happens in a type-safe way. Claims are commonly made in the popular literature that this is indeed the case. However, this is not immediately clear from the specification.

This thesis is an investigation into the different mechanisms that CORBA specifies to support remote operation invocations and a discussion of whether these mechanisms preserve type-safety for cross-boundary operation invocations. Successively, the object model, the type system, the architecture and the development process are reviewed. This is followed by a detailed investigation into the communications protocol used by CORBA, the server-side request dispatching mechanism and client-side operation invocation mechanisms. Conclusions drawn from these investigations are used to discuss type equivalence and the issues around interface evolution.

Acknowledgements

This thesis would not have been written without the support of many people. First of all, I would like to thank all current and former members of the Rapids research group at Glasgow University and the HIPPO and Smartlab research groups at Strathclyde University for providing me with a congenial working environment. In particular I would like to extend my appreciation to Prof. M. Atkinson for the warm welcome when I first arrived in Glasgow and Prof. A. McGettrick when I returned there.

Prof. Paddy Nixon must be thanked for his leniency in letting me get away with 'things' during the writing of this thesis (especially the missing of deadlines) and for proofreading parts of the document. Both were well appreciated.

I would like to thank Fabio Simeone for the many useful (and enjoyable!) discussions. A lot of pieces in this jigsaw were formed during these meetings.

Special thanks go to Prof. Richard Connor, my supervisor, for the sound technical advice and the heaps of confidence. I would also like to thank Richard for telling me that obtaining a master's degree is a piece of cake. It stopped me from giving up when it looked like I would never finish it. But at the same time I would like to scold him for this, as it probably was not the smartest of things to say to someone who only knows last-minute panic.

Special thanks also go to Claudie for her love and for coping with the last-minute stress. I promise, this was the last time ;)

Above all, I would like to thank my parents for their support throughout the years and for the opportunity to study abroad. It is only natural that I dedicate this work to them.

David Lievens.

Table of Content

1 Introduction 5
1.1 Middleware 5
1.2 Common Object Request Broker Architecture 6
1.3 Preface to the Thesis 7
2 An Overview of CORBA 8
2.1 Object Model 8
2.2 Type System 11
2.2.1 Object Types 11
2.2.2 Non-Object Types 11
2.2.3 Interface Definition Language 12
2.3 Architecture 13
2.4 Development Process 14
2.4.1 Establishing the interfaces 15
2.4.2 Server-side Development 16
2.4.3 Client-side Development 17
2.4.4 Change 18
2.5 Dynamic Invocations and Interface Repository 18
2.6 Example 19
2.6.1 Introducing the Example 19
2.6.2 Specification of interfaces 20
2.6.2.1 Compilation of interfaces 21
2.6.3 Server-Side Development 22
2.6.3.1 Implementation of interfaces 22
2.6.3.2 Server program 23
2.6.4 Client-Side Development 24
2.6.4.1 Static Invocation Interface 24
2.6.4.2 Dynamic Invocation Interface 24
3 Preserving Strong Typing 27
3.1 Strong Typing 27
3.2 Communications Protocol 29

3.2.1 General Inter-ORB Protocol ... 29
3.2.2 Message Formats ... 32
3.2.2.1 Request Message ... 33
3.2.2.2 Reply Message ... 34
3.2.3 Common Data Representation ... 35
3.2.4 Interoperable Object Reference ... 36
3.2.5 RepositoryId ... 38
3.3 Server-side Request Dispatching ... 39
3.4 Client-side Operation Invocation ... 42
3.4.1 Static Operation Invocation ... 42
3.4.2 Dynamic Operation Invocation ... 46
3.5 Strong Typing in CORBA ... 46
3.6 Type Equivalence ... 47
3.6.1 Type Equivalence in CORBA ... 48
3.7 Interface Evolution ... 50
3.7.1 A very short note on DCOM ... 53
4 Conclusions ... 54
References ... 57

1 Introduction

1.1 Middleware

Modern distributed computing platforms are inherently heterogeneous [Em00, HV99]: mainframes handle transactional database access, workstations perform computing intensive tasks and personal computers run a host of office automation tools. Additionally, there might be dedicated systems present for controlling telephony systems and specialised measurement equipment.

Middleware provides abstractions for distributed object oriented programming to allow application developers to seamlessly integrate diverse applications into heterogeneous distributed systems. Middleware can be seen as a software bus - analogous to a hardware bus- on which software components are plugged (see Figure 1). It defines the wiring and the protocol for interaction, allowing well-behaving software components to interoperate.

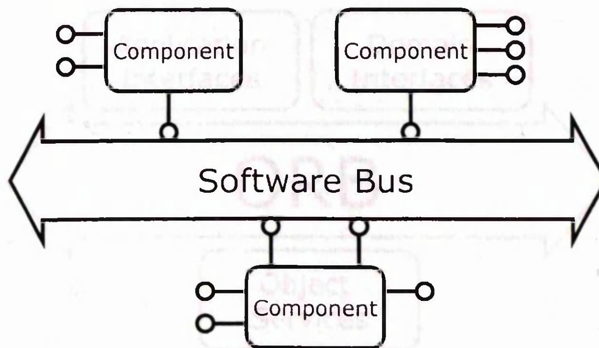

Figure 1 The Software Bus

Wiring standards are not enough though [Szy]. If no domain specific semantic issues are included in the standard, significantly more effort needs to be put in making connected components interoperate. Middleware would usually try to address this by offering a number of semantic layers, ranging from low-level services, essential to the working of almost all corporate applications, to high-level domain specific frameworks designed specifically for one class of applications.

There is a wide range of middleware products available on the market, ranging from message-oriented middleware over transactional middleware to object oriented

middleware. In this thesis, we are only considering the Common Object Request Broker Architecture, which is a specific kind of object oriented middleware.

1.2 Common Object Request Broker Architecture

The Object Management Group (OMG) is a consortium of more than 800 vendors who banded together to propose a middleware specification. The specification created by the OMG is the Object Management Architecture (OMA), of which the Common Object Request Broker Architecture (CORBA) is a fundamental part.

The OMA specification aims to create a framework in which a complete corporate information system can be developed (cf. Figure 2). It specifies object services (CORBAServices) such as a naming service, events service, transactions service, etc. It also specifies common facilities (CORBAFacilities): horizontal end-user-oriented facilities that are applicable to most application domains, such as an OpenDoc-based Distributed Document facility. And it specifies a number of domain interfaces for application domains such as finance, healthcare, manufacturing, telecom, etc.

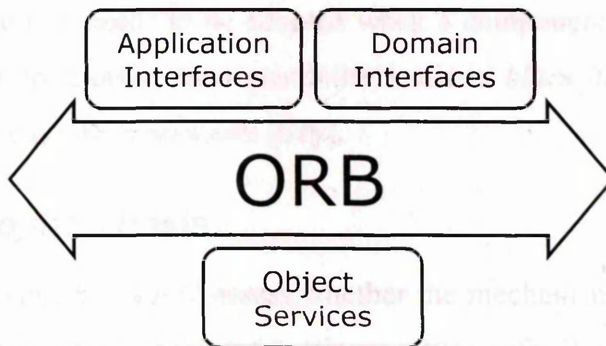

Figure 2 The OMA Interface Categories

The central specification that enables all the previous things to form a coherent framework is CORBA. The CORBA specification outlines an infrastructure allowing objects to communicate, independent of the specific platforms and techniques used to implement the addressed objects. It allows a great deal of abstraction over the difficulties of inter-process, cross-platform and even cross-language application development.

The CORBA specification is very complex and is even said to contain contradictions [BS]. The core of the specification covers almost a thousand pages and is

complemented by a number of language mappings. CORBA is a specification and can be implemented by different vendors. Among the information contained in the specification, we can identify a number of themes. The first theme is about the CORBA *data model* [chapter 1]. The data model discusses the central modelling primitives that are offered by CORBA. A second theme is the specification of additional services that do not fit with the core data model, but for which *portability* is important and can therefore not be left entirely to different ORB implementers [chapters 3,4,5,6,7,8,9,10 and 11]. A third theme revolves around *interoperability* issues [12,13,14,15,16,17,18,19 and 20].

Both interoperability and portability help to reduce dependencies on the environment and therefore reduce the risk of locking in a company with a single ORB vendor. Interoperability does so by maximising the amount of components that can talk to each other. When interoperability between two ORB implementations is established, components connected to one system can also talk to components connected to the other system. Portability reduces the dependency on the environment by minimising the amount of code that needs to be adapted when a component is moved from one middleware system to another. Interoperability leads to *black box* reuse, portability leads to *white box* reuse of components [Szy].

1.3 Preface to the Thesis

In the following chapters, we will assess whether the mechanisms CORBA provides to enable inter-address space object interaction are type-safe. We review successively the object model, the type system, the architecture and the development process. We then continue with a thorough discussion of the communications protocol, the server-side request dispatching mechanism and the different client-side operation invocation mechanisms. At that point we will be able to discuss type-safety issues. We conclude with a discussion of type equivalence and interface evolution.

2 An Overview of CORBA

In this chapter, we give a high-level overview of CORBA to provide a framework for the discussion that follows. We start by reviewing the object model, the type system and the architecture of CORBA. Discussing the object model will give insight in the logical abstractions that CORBA offers to facilitate construction of distributed applications. Analysing the architecture will then show how these logical abstractions are supported by an implementation. Introducing CORBA in this manner avoids issues related to particular request broker implementations or language-bindings. To be able to follow the discussion in the next chapters, it is equally important to be familiar with how CORBA applications are constructed. We give an overview of the development process and conclude by demonstrating it with a small but representative example.

2.1 Object Model

The *data model* of a system is the set of modelling primitives that it offers. Discussing a system in terms of its data model provides abstraction over the concrete implementation of the system. A well-known data model is the relational data model. It allows modelling of information organisation, information storage and information retrieval in terms of tables, columns, rows, primary keys, foreign keys, constraints, etc. A large number of implementations exist for the relational model in the form of Relational Database Management Systems such as Oracle, Microsoft SQL Server and others, each of which has distinctive additional properties and features.

When a data model is organised around the generic concept of an object –as is the case with CORBA- it is usually called an *object model*. The CORBA object model contains primitives for modelling peer-to-peer object interaction in a distributed computing environment. The specification assumes that any implementation of the object model relies on one or more programming languages, which we will denote by the term implementation languages.

An overview of the different modelling concepts:

- *Object* – identifiable, encapsulated entity with an associated state that offers a set of services that can be requested by a client. A *client* of a service is any entity capable of requesting the service. The state is not directly accessible; it can only be queried and modified via the services the object offers.

  Objects have a lifetime. There is no special mechanism to create or destroy objects; instead, new objects are created and existing objects destroyed as a side effect of issuing regular requests on existing objects. How the bootstrapping problem is solved, is not addressed by the object model. The outcome of object creation is an object reference that denotes the new object (see below).

  CORBA defines[1] a type system. Objects are instances of *interface types*. The interface (type) specifies what services an object offers. An interface consists of a set of operation signatures. Each operation signature consists of an identifier, an ordered[2] list of input parameters and an ordered list of output parameters (including a single return value). The parameters are also typed. We discuss the type system in more detail in Section 2.2.

- *Object Implementation* or *servant* – provides an object's state and an implementation for its operations. Servants have no identity that can be observed by the client. Servants must be provided using mechanisms that lie outside the scope of the CORBA specification. They provide a complete separation of interface and implementation. Two objects of the same object type can have completely different implementations.

  There is not necessarily a one-to-one mapping between objects and servants. One servant can implement multiple objects and multiple servants can implement one object. Additionally, the lifetime of objects and servants is independent. An object can be implemented by multiple servants over time. This provides the possibility for fault-tolerance or for increasing the performance of a system by doing load balancing.

---

[1] At least partially, see below.
[2] The specification explicitly states that request parameters are identified by their position [Core p1-3].

- *Object Reference* – value that reliably denotes a particular object. Multiple copies of an object reference -denoting the same object- can exist. A client that wants to request services of an object needs to obtain a reference for that object. In CORBA, object references are opaque to the client application programmer. There is no standardised API to inspect the content. Therefore, object references provide *location transparency*, i.e. abstract over the physical location of an object.

- *Request* – interaction between two objects. The source of the interaction is called client and the destination server. The information associated with a request consists of a target object, an operation identifier (which is the operation's name[3]) and zero or more (actual) parameters. A request causes a service to be performed on behalf of the client. One possible outcome of performing a service is the availability of any results defined for that request. If an abnormal condition occurs during the execution of a request, an *exception* is made available instead.

  CORBA has three invocation modes: *synchronous*, *deferred synchronous* and *one-way*. When a client issues a synchronous request on an object, it blocks while it waits for the results to become available. When it issues a deferred synchronous request, it continues processing and can then later poll or block-wait for the response. When it issues a one-way request, it continues processing and it does not expect any results to become available. This last invocation mode assumes best-effort execution semantics. It allows requests to be silently dropped in case of network congestion or other resource shortages. The first two invocation modes assume at-most-once execution successfully, i.e. if an operation request returns successfully, it was performed exactly once; if it returns an exception indication, it was performed at-most-once.

- *Exception* – indication that an operation request was not performed successfully. CORBA defines two kinds of exceptions: system exceptions and user exceptions. System exceptions are automatically raised when failures are detected such as

[3] The specification explicitly states that operations are identified by their name [Core p1-6]. Therefore, operation overloading is not allowed when defining interfaces in IDL (cf. infra).

disconnected network connections, etc. User exceptions are raised by the target object, when a request could violate the object's integrity. An exception may be accompanied by additional, exception-specific information.

2.2 *Type System*

CORBA partially defines a type system. For non-object types it only indicates the concepts, but for object types it is more specific.

2.2.1 Object Types

An object type or interface type contains a set of operation signatures. Each operation signature consists of an identifier, an ordered list of input parameters and an ordered list of output parameters. Each parameter has a name and a type. The parameter-passing is always by value (although the value could be a reference). The only operations applicable to an instance of an object type are the operations specified.

Each interface has a unique identity that discriminates it from all other interfaces (even though their syntactic representations may be identical). Subtype relationships between interfaces can exist, but have to be established explicitly. This may avoid some unwanted connections between object types, but comes at the cost of flexibility[BW].

2.2.2 Non-Object Types

CORBA defines a number of *basic types*; a representative subset is shown in Table 1. For each basic type, it specifies a range, a minimum size and an encoding (e.g. 2-complement, ISO Latin-1, etc.)[4]. The different language mappings [OMGa, OMGb] complementing the core specification specify additionally a mapping from each basic type to a language-specific type (e.g. in Java, the long integer type is mapped on the built-in type `long`). Neither the language mapping nor the core specification define any operations on basic types; what operations are defined depend on the semantics of the mapping type in the implementation language that is used and may therefore vary.

---

[4] The size and the encoding are specified in the chapter on the data model. The range is specified in the chapter on IDL. It seems that it would have been better just to specfy the range in the chapter on the data model and leave the size and encoding to the specification of the data representation.

Type	Range	Size	Encoding
integer	-2^{15} to $2^{15}-1$	≥16 bits	2-complement
long integer	-2^{31} to $2^{31}-1$	≥32 bits	2-complement
unsigned integer	0 to $2^{16}-1$	≥16 bits	plain binary
unsigned long integer	0 to $2^{32}-1$	≥32 bits	plain binary
floating point	*specified by encoding*	≥32 bits	IEEE single-precision
double floating point	*specified by encoding*	≥64 bits	IEEE double-precision
character	*specified by encoding*	≥8 bits	ISO Latin-1
string	finite strings of length $\leq 2^{32}-1$	variable-length	ISO Latin-1
boolean	{true, false}	unspecified	unspecified
octet	0 to 255	≥8 bits	n/a
infinite Union	n/a	variable-length	n/a

Table 1 CORBA Basic Types

CORBA also defines a number of *type constructors*. It defines a record type constructor, a discriminated union type constructor and sequence and array type constructors. Again, no operations are formally defined on these type constructors. But it is clear that the only operation applicable to a record type constructor is field selection. However, the exact operations available depend on the type constructors on which they are mapped in the implementation languages. The record type constructor applied to a set of labels and a set of types is mapped in Java onto a class containing attributes representing the different (name, value) pairs and therefore.

Applying a type constructor to a (number of) basic type(s) yields a *complex type*. Complex types must usually be named, as anonymous types are only allowed in a limited number of cases. Unlike what was the case for object types, it is not possible to declare inheritance (substitutability) relationships between complex types.

2.2.3 Interface Definition Language

CORBA provides syntax for writing down type definitions in a language-neutral way. This syntax is called the (OMG) Interface Definition Language (IDL). It resembles the syntax of C++ and Java. A discussion of the exact syntax and semantics can be found in [Core chapter 3, HV]. An example follows later in this chapter (see Section 2.6.2)

2.3 Architecture

In the last section, we have given an overview of the logical abstractions that CORBA offers. This section focuses on the different components that are involved in implementing that logical picture. Figure 3 shows the components of a CORBA-based application that are involved in executing a request. In the picture, it is assumed that some client object issues a request on some target object located in a different address space and that an object request broker is present at both sides.

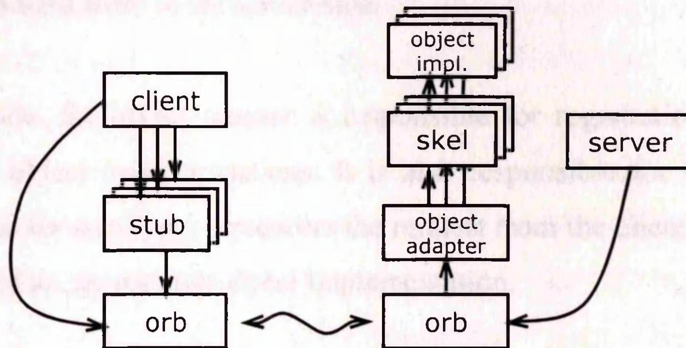

Figure 3 The CORBA Architecture

The Object Request Broker is the central component of the CORBA architecture. An ORB receives a request to invoke an operation from the stub of a client object and forwards that request transparently to the request broker at the server-side. In particular, the client ORB locates the server ORB based on the addressing information contained in an object reference, transmits the request parameters to the server and returns the request results to the client.

An ORB is usually implemented in the form of a library; the specification [Core chapter 4] specifies the interface of an ORB. This means that it is language-specific; for each component (either client application or object implementation) written in a different language, a different ORB needs to be present. A client application that wants to invoke remote operations, need to initialise the ORB first. Once this is done, the ORB is normally only accessed through a stub. A stub is usually generated automatically from a set of type declarations. The tool that generates stubs is an integral part of any ORB implementation.

A stub is an object (not necessarily in the object-oriented meaning) of the implementation language in which the server is written. The client can access the stub in the same way as it would access other language objects (again, not necessarily in the object-oriented meaning). As the stub acts as a local stand-in for a remote object, this provides access transparency [Em00] for the client; i.e. the client cannot discriminate between accessing a local object and accessing a remote object. The stub marshals the parameters of the operation that is invoked by the client application, and uses the ORB to send them to the server-side.

At the server-side, the object adapter is responsible for registration, activation and deactivation of object implementations. It is also responsible for the generation of object references for an object. It receives the request from the client and dispatches it to the skeleton of an appropriate object implementation.

The skeleton demarshals the parameters which were marshalled by the stub. It then invokes the requested operation on the target object implementation. As was the case for the stub, the skeleton is an entity belonging to the language in which the object implementation has been written. This shields the implementer from any low-level networking issues.

It should be noted that this is only a conceptual picture and serves mainly as a framework for discussion. The different components listed here all have a set of logical responsibilities for supporting request flows. However, in some request broker implementations, none of these components may have a physical counterpart. The main reason for discussing them is to create reference framework that can be used when discussing other services that hook into or utilise some functionality that is specified here.

2.4 Development Process

In the previous sections, we have introduced the CORBA object model its type system and we have briefly discussed its conceptual architecture. For the discussion to follow it is also important to understand the steps that application developers go through when constructing CORBA applications.

Like the first part of any application that is being developed using an object-oriented methodology, the first step in developing a CORBA application is establishing the interfaces. Interfaces are specified using a special purpose specification language, called the Interface Definition Language (IDL). Once the interfaces are specified, they are translated into type declarations[5] in the language(s) of choice of the interface implementers and interface consumers. The interface implementers are then responsible for providing an implementation for these types and the interface consumers write applications that employ instances of these types. Also, both the implementer and consumer need to write some code to activate the request broker and access the request broke libraries.

A sketch of the different steps in the development process can be found in Figure 4. As before, we simplify the discussion to a single consumer and implementer, which we will refer to as being the client-side and server-side respectively. Next, we discuss the different tasks in the development process in some more detail.

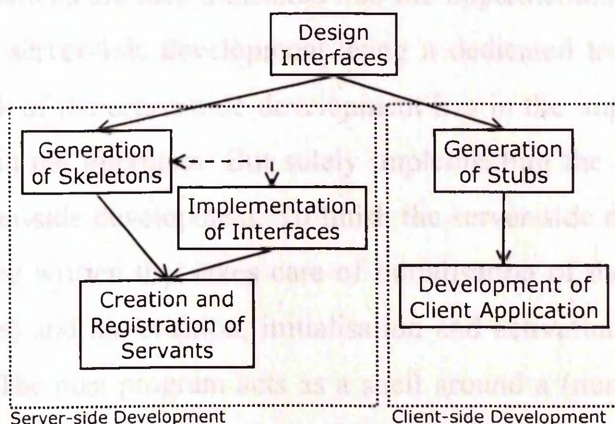

Figure 4 Development Process of CORBA Applications

2.4.1 Establishing the interfaces

Similar to what happens in any object-oriented development process, one has to start with establishing the interfaces when developing a CORBA application. Distributed computing, however, has some additional characteristics that need to be taken into account: *message latency*, *network partitioning* and *partial failure*. CORBA cannot

[5] In case of dynamically typed languages that have no explicit type declarations, the IDL interface specifications may be translated in an outline of the implementation.

entirely hide the effects that these characteristics have on applications. Messaging latency has a potentially profound influence on the application's performance and it is important to keep this in mind during interface design. Network partitioning may deny uniform addressing of network nodes, thereby preventing the location transparency offered by CORBA object references[6]. Finally, partial failure demands for more sophisticated failure handling strategies than are normally employed in monolithic software applications. And, as exceptions need to be specified in the signature of the methods that can generate them, failure handling needs to be taken into account during interface design.

Once the interfaces are fixed, client and server-side development can happen in parallel. We assume in this discussion physically separated client and server platforms.

2.4.2 Server-side Development

Server-side development starts with the acquisition of the IDL type specifications. These type specifications are then translated into the implementation language that is being used for the server-side development using a dedicated translation tool (IDL compiler). The bulk of the server-side development lies in the implementation of the services specified in the interfaces. But solely implementing the interfaces does not suffice for the server-side development. To finish the server-side development, a host program needs to be written that takes care of initialisation of the ORB, creation of the object adapter(s) and the creation, initialisation and activation of (at least some) CORBA objects[7]. The host program acts as a shell around a (number of) servant(s), providing an entry point for the CORBA application. We refer to the example in the next section and the appendices for more information on this part of the application.

The implementation of the interfaces needs to be done in a programming language for which a language mapping exists[8] (and for which an ORB implementation is available). Note that this language does not need to be object-oriented. Although the

---

[6] Bridge-objects may have to be positioned on the computers sitting between two networks to forward requests to objects in he *other* network.

[7] CORBA objects are usually created as the result of requests, but for bootstrapping reasons, some must be instantiated by other means.

[8] I.e. a language in which the IDL interface declarations can be translated.

CORBA object model contains the notions of object and interface type, the architecture is flexible enough to support non object-oriented programming paradigms. In fact, it suffices that some sort of parameterised expression abstraction [Sch90] is available for a mapping to be conceivable.

One of the important goals of CORBA is to support legacy applications; the implementation of the interfaces does not necessarily has to happen from scratch. But even when the services are already available in some form, it will generally be necessary to provide at least a wrapper that forwards incoming requests to the legacy application[9].

Although CORBA has a facility to dynamically host objects (called the Dynamic Skeleton Interface or DSI), in this document we are always assuming that servers have static knowledge about the objects they host. In the light of our previous discussion of middleware and componentware, this is a natural assumption.

2.4.3 Client-side Development

As was the case for the server-side development, the client-side development starts with the acquisition the IDL type specifications and the translation of these specifications into a development language of choice. The application developer can then use these type declarations in his program in the same way as *normal* type declarations. This provides *access transparency* [Em00] for instances of CORBA types.

Clients do not handle CORBA objects directly, they use object references. An object reference uniquely identifies a CORBA object and provides *location transparency* [Em00], i.e. the client does not have to know the physical location of the server-object in order to access it. A client that wants to utilise a certain CORBA object, needs to obtain a reference to this object. References can be obtained in a number of ways (cf. Section 3.4.1). Object references are mapped differently in different languages, but

[9] Note that the wrapper must then be written in a programming language for which a language mapping exist, but no such constraint applies to the legacy application, which may be written in any language. In fact, this is probably the reason why the commercially available Java ORBs are so popular. The wrapper can be written in Java and the legacy application can be accessed through Java's Native Interface.

the type of the object reference always conforms to the type resulting from the interface of the object it refers to and therefore, they may be treated as if they were the objects themselves.

2.4.4 Change

If the interfaces need change at any point, then the IDL specification needs to be adapted first. The updated IDL specifications must then be distributed to clients and servers. They must compile these new specifications and incorporate the resulting type declarations (e.g. modify an operation invocation to reflect the changes in the interface). Finally, the whole application must be redeployed. So, when a change is made to an interface, all the clients of that interface must be redeployed.

2.5 Dynamic Invocations and Interface Repository

Sometimes, it is necessary to be able to develop clients that do not have compile-time access to the interface definitions of the services they want to use. This may be because of the fact that the interface definitions are not available when the client program is being developed. Or it may be because the application is generic and it is impractical to incorporate all the interface definitions that it must be able to handle[10]. Examples include gateways, bridges, object browsers, distributed debuggers, etc.

To cater for the needs of such applications, CORBA provides the *Dynamic Invocation Interface* (DII) and the *Interface Repository* (IFR). The Dynamic Invocation Interface allows dynamic creation and invocation of requests on an object. A client can use this interface to send a request to a target object and will obtain the same semantics with it as a client using stubs generated from the type specification. A server cannot discriminate between requests issued by a client that uses the DII or a client that uses a stub.

Accessing services using stubs can only happen in a synchronous way. This is because stubs have been introduced to provide access transparency and method invocation happens in a synchronous way in most programming. The DII provides the possibility for deferred synchronous and one-way execution access to services.

[10] When all the interfaces that must be handled have a common defined supertype, it is only necessary to have access to the declaration of this supertype.

The Interface Repository (IFR) is a component that provides persistent storage of interface definitions. It is a regular CORBA object and can be queried by a client to obtain type information at runtime. Using this information, a client can then employ the DII to issue a request. The IFR is usually populated by the IDL compilers provided with ORB implementations.

Using these two mechanisms, a client can obtain a reference to an object on which it has no (static) knowledge, enquire about the interface of that object using the IFR and use the knowledge obtained from the IFR to issue requests using the DII.

2.6 Example

We have discussed the object model of CORBA, a set of concepts and modelling primitives that can be used to model a domain of interest. We have also discussed the architecture of CORBA and we have seen how the different pieces work together to implement the logical abstractions provided by the object model. Finally, we have outlined the development process that needs to be followed when constructing a CORBA application. It is time to demonstrate all this by a small example.

Note that, although we have occasionally included code fragments in this section, no understanding of the concrete syntax is necessary to follow the discussion in the following chapters.

2.6.1 Introducing the Example

Imagine a simple bank account application with as its main abstraction a normal savings account. The basic functionality consists of querying the balance of the account, depositing money into the account and withdrawing money from the account. For the sake of the argument we choose the currency of the account to be Belgian Franks. Belgian Franks do not go beyond the decimal point. Therefore, deposits and withdrawals can only happen by integer amounts. In other words, it is possible to deposit 875 Bfr. but not 1000,5 Bfr. Let us further assume that the account has no overdraft facility. The maximum amount of money that can be withdrawn at any point is the full balance of the account at that point. If it is attempted to withdraw more money than is actually present in the account, just the full balance is returned;

no error is raised. Each account has an accountholder, i.e. the person who owns the money that is in the account. For simplicity, we assume that the name of the accountholder uniquely identifies an account.

Operations on an account can be performed by many different entities, such as automatic teller machines, terminals in branches, other software components (e.g. from a web-banking application) and telephony systems (e.g. phone banking). Management of accounts happens through a separate mechanism. Management functionality includes opening of new accounts, closing of existing accounts and returning existing accounts on request.

Although this is a small, restricted example, it does demonstrate the possibilities of CORBA quite well. Great deal of heterogeneity. Legacy applications. Easily decomposable in a set of interfaces.

2.6.2 Specification of interfaces

Another nice property of the example is that it is easy to decompose the specified functionality into a set of interfaces. Expressing the aforementioned specification in IDL results in two interfaces: `Account` and `AccountManager`, which we have grouped together in a module called `BankApplication`. Modules serve as a namespace construct in IDL. The `Account` interface contains methods to query and update the balance of a savings account. The type of the parameters is locally defined and is called `Currency`; it is defined as an alias to `unsigned long`. This suffices because Belgian Franks go not beyond the decimal point and we have agreed that the account has no overdraft facility. The `AccountManager` interface contains functionality to manage accounts, such as creating and destroying accounts and looking an account up. As mentioned before, we include the IDL source for the interested reader; no thorough understanding is needed to follow the argument. Like the module construct, the prefix pragma also serves for namespace purposes. We will explain its specific purpose later (cf. Section 3.2.5).

```
// OMG IDL
#pragma prefix "hippo.dcs.gla.ac.uk"

module BankApplication
{
```

```
typedef unsigned long Currency;

interface AccountManager
{
    Account open (in String name);
    void    close(in String name);

  Account get(in String name);
};

interface Account
{
    readonly attribute String accountHolder;
    readonly attribute Currency balance;

    void deposit(in Currency amount);
    Currency withdraw(in Currency amount);
};
};
```

2.6.2.1 Compilation of interfaces

One of the things that an actual CORBA implementation must provide is an IDL compiler. An IDL compiler takes an IDL specification and does a language-specific mapping of the types specified. It generates stub and skeleton code that handle most of the low-level stuff, such as marshalling data, invoking the ORB API, etc. Compiling the above type definitions with the ORBacus 4.0 IDL-to-Java compiler [OOC] yields the following files:

```
Account.java
AccountOperations.java
AccountHelper.java
AccountHolder.java
AccountPOA.java
_AccountStub.java
Account_impl.java

AccountManager.java
AccountManagerOperations.java
AccountManagerHelper.java
AccountManagerHolder.java
AccountManagerPOA.java
_AccountManagerStub.java
AccountManager_impl.java
```

CurrencyHelper.java

The files that are generated differ from language to language and for some language mappings even from ORB implementation to ORB implementation. In general, for most languages, there are different files generated for *the interface types* (Account.java, AccountOperations.java, AccountManager.java, AccountManagerOperations.java), for *the stubs* (_AccountStub.java, _AccountManagerStub.java) for *the skeletons* (AccountPOA.java, AccountHelper.java, AccountManagerPOA.java, AccountManagerHelper.java). Sometimes additional files are necessary to assist in translating the CORBA object model into language specific constructs. Java can, for example, not directly support inout parameters. Therefore, additional classes need to be introduced (AccountHolder.java, AccountManagerHolder.java).

The files in which the different interface types are declared must be generated for both the client and server (possibly in different languages, using different IDL compilers). The stubs must only be generated at the client side; the skeletons only at the server-side.

2.6.3 Server-Side Development

Once the translation of the IDL specification into the target language has been completed, the server-side development can be started. As we have seen in the section on the development process, this consists of two stages: implementation of the interfaces and construction of a server program.

2.6.3.1 Implementation of interfaces

For object-oriented implementation languages, it is very common that a matching abstract base class is generated from each IDL interface. The application programmer must then subclass this class to provide an implementation for the services specified in the IDL specification. In this case, partial implementations have been generated for the interface implementations: Account_impl.java and AccountManager _impl.java. The application programmer should build on these files to provide

the implementation for the different methods. The complete implementation of the Account interface can be found in the appendices. For the interested reader, we have included an outline of the implementation class.

```
//Java
//outline of the part automatically generated by the
//IDL-to-Java compiler

package uk.ac.gla.dcs.hippo.BankApplication;

//IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0
public class Account_impl extends AccountPOA
{
  //automatically generated constructors omitted

//IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/account
Holder:1.0
  public String accountHolder(){
   //implementation to be provided
   }

   //signatures of other methods omitted

   // IDL:hippo.dcs.gla.ac.uk/BankApplication/ ↵
Account/withdraw:1.0
   public int withdraw(int amount){
    // implementation to be provided
    }
}
```

2.6.3.2 Server program

As we have seen in Section 2.4.2, the implementer must also provide a server program that is responsible for instantiating and registering servants of objects with the ORB. This program acts as the administrative glue that glues CORBA objects to a real operating system environment. We have omitted the listing of the program here, the complete source code can be found in the appendices.

2.6.4 Client-Side Development

We demonstrate the client-side development both using the Static Invocation Interface and the Dynamic Invocation Interface.

2.6.4.1 Static Invocation Interface

Again, the IDL specifications must be compiled into type declarations and stubs. Once this is done, the client program can be developed. As has been explained above, the client program can access remote services in two ways: using the Static Invocation Interface and using the Dynamic Invocation Interface.

For this example, we have developed a simple stand-alone client. Because it is a stand-alone application, it first has to initialise the ORB and obtain the initial references. Initial references are necessary for bootstrapping reasons and are usually obtained by means of a special primitive of the ORB. Once this is done, a reference must be obtained to the service we are interested in: an instance of the Account interface. We can then invoke operations on this object in the same way we would invoke operations on a regular Java object. We have omitted most of the code of the example here, only some example method invocations are listed. The complete code of the example client can be found in the appendices.

```
//Java
    AccountManager m = ...
    Account myAccount = m.open("david");
    MyAccount.deposit(1000);
    //...
```

2.6.4.2 Dynamic Invocation Interface

With the Dynamic Invocation Interface, the IDL interface declarations of the server are not necessary. It suffices that we know the name of the operations we want to invoke and their signatures. Otherwise we have to obtain this information from the IFR (not shown here). When using the DII, requests are CORBA objects themselves.

To invoke an operation on an object with the DII, the following steps must be followed:

1. Construction of a `Request` object for the operation:

To construct a Request object r, one must have first obtained a reference or to the target object. On this object reference, the _create_request() operation is applied (which is defined in the interface of Object). This operation takes - among other things- an operation identifier and a list[11] of arguments to the operation and returns a Request object.

2. Invocation of the request

   Once a Request object is created and populated with the different request parameters, the operation can be requested by a call to the invoke() method of the object. This method calls the ORB, which performs method resolution and invokes an appropriate method. The result (either the return values or an exception) of issuing the request is placed in a dedicated field of the Request object.

   The DII also provides the possibility of issuing requests that have deferred synchronous or one-way invocation semantics. In these cases, the request should be made using the send() method and results (in case of deferred synchronous calls) can be obtained via the poll_response() method.

The code for the two lines that were shown in the section on the SII:

```
//Java
Object o = ... //obtain object reference
NVList myArguments = ... //generate ordered list of args
Request r =
  o._create_request(..., "open", myArguments, ...);
r.invoke();

Object account = ... //get obj ref out of return val
NVList otherArgs = ...
Request r2 =
  o._create_request(..., "deposit", myArguments, ...);
... // get results out of NVList
```

[11] The datastructure containing the arguments is standardised. See [7-2] for a complete discussion.

3 Preserving Strong Typing

In the previous chapter, we have discussed the CORBA object model and its type system. Objects have interfaces and clients can only issue requests that conform to these interfaces. We have also seen that the CORBA specification consists of more just than an object model. It also takes portability and interoperability issues into account. It does so by defining a number of APIs and some operational mechanisms that different ORB implementations must support. On page 13-17, the specification [Core] states that it allows the preservation of strong typing. Additionally, a common claim in the popular literature is that CORBA allows static typing through the use of stubs and skeletons [HV, Ba96, Lew00, RG]. This chapter considers the typing issues in CORBA in detail and comments from this viewpoint on important issues such as type equivalence and interface evolution.

3.1 Strong Typing

In this treatment, we consider a typed programming language to be *strongly type-checked* or *strongly typed* when it allows the manipulation of values only through operations that are defined on the type of that value. Types and their operations are specified in the type system that is associated with the programming language. Enforcing strong typing is important. Types perform a modelling role: applying an operation that is not defined for a type to an instance of that type violates the modelling intention of that type[16].

Checking whether undefined manipulations take place can happen at any time before the execution of those operations. Strong typing can be achieved by guarding all manipulations with an explicit check at run-time. Alternatively, strong typing can be achieved through analysis of the source code. In the latter case, a type must be associated with every denotation in the program text, in order to allow unsound manipulations to be mechanically spotted at compile-time.

In general, the earlier an error is detected, the less harmful it might be. Therefore, it is desirable to do type checking as early as possible, ideally at compile time. However,

---

[16] Or the modelling has not captured the concept satisfactorily.

3 Preserving Strong Typing

In the previous chapter, we have discussed the CORBA object model and its type system. Objects have interfaces and clients can only issue requests that conform to these interfaces. We have also seen that the CORBA specification consists of more just than an object model. It also takes portability and interoperability issues into account. It does so by defining a number of APIs and some operational mechanisms that different ORB implementations must support. On page 13-17, the specification [Core] states that it allows the preservation of strong typing. Additionally, a common claim in the popular literature is that CORBA allows static typing through the use of stubs and skeletons [HV, Em96, Em00, Rit]. This chapter considers the typing issues in CORBA in detail and elaborates from this viewpoint on important issues such as type equivalence and interface evolution.

3.1 Strong Typing

In this treatment, we consider a typed programming language to be *strongly type-checked* or *strongly typed* when it allows the manipulation of values only through operations that are defined on the type of that value. Types and their operations are specified in the type system that is associated with the programming language. Enforcing strong typing is important. Types perform a modelling role; applying an operation that is not defined for a type on an instance of that type violates the modelling intention of that type[12].

Checking whether undefined manipulations take place can happen at any time before the execution of those operations. Strong typing can be achieved by guarding all manipulations with an explicit check at run-time. Alternatively, strong typing can be achieved through analysis of the source code. In the latter case, a type must be associated with every denotation in the program text, in order to allow unsound manipulations to be mechanically spotted at compile-time.

In general, the earlier an error is detected, the less harmful it might be. Therefore, it is desirable to do type-checking as early as possible, ideally at compile time. However,

[12] Or the modelling has not captured the concept satisfactorily.

it is not always possible to do all checking at compile time. When values only become available at run-time[13], as is the case with distributed and persistent data, the earliest time possible to check is when the value actually becomes available.

However, even in these cases, in languages in which denotations carry type annotation, *most* checks can be performed at compile time. The only dynamic checks that are needed are the checks to assess whether the values that become available at run-time conform to the types specified by the programmer for the variables containing them. All manipulations of variables can be statically checked with respect to the type annotations of those variables.

In the literature, a dichotomous distinction is usually made between *statically* and *dynamically* checked languages. Statically checked languages being languages that achieve strong typing through analysis of the source code and dynamically checked languages being languages that employ explicit run-time checks. However, most so-called statically typed languages incorporate features that cannot be entirely checked at compile-time and most so-called dynamically typed languages implement compile-time optimisations to eliminate some run-time checks. Therefore, it would be more precise to categorise them into *mostly statically* and *mostly dynamically* checked languages, but for the remainder of this document we will adhere to the terms as they are typically used in the literature.

CORBA is not a programming language in its own right; it extends programming languages to enable access to objects in different address spaces. The extension usually comes in the form of a library. Therefore, strictly speaking, neither the syntax nor the semantics of the host programming language is changed. And, thus also not whether the language is strongly typed or not. Yet CORBA allows the programmer to write code that results in the invocation of methods on remote objects. These objects may be implemented using a different programming language with different semantics and a different type system. The mechanisms it specifies to make this

[13] The alternative is that the program itself creates the value. In that case static checking is –in general- possible.

possible can be seen as an extension of the implementation language's run-time system.

When assessing strong typing in such 'enhanced' programming languages, we are interested in whether undefined operations can be invoked. We have to make a distinction between the CORBA interface type and all CORBA non-interface types (i.e. the basic types plus the constructed types). Instances of the latter category are always manipulated within a single language framework; it is only possible to transmit them from one environment to another as part of the parameters or return value of an operation invocation, it is not possible to invoke operations on them from within a different environment. Therefore, it depends on the algebra of the implementation languages used whether or not those values can be manipulated in ill-defined ways.

Manipulations of instances of object types cross the boundaries between client and server. CORBA specifies a number of mechanisms to support these manipulations and relies on the semantics of the implementation languages to provide semantics for the manipulations that happen inside a particular environment. It is these support mechanisms that will make the subject of the investigation into strong typing for the remainder of this chapter.

We start by looking into the communications protocol that CORBA employs. We then go on to look into the server-side request dispatching mechanism and the client-side operation invocation mechanisms. Having studied these in detail, we will be able to assess whether CORBA is strongly typed.

3.2 Communications Protocol

3.2.1 General Inter-ORB Protocol

Middleware specifications, such as CORBA, leave room for varying implementations. This is important from a commercial point of view, as vendors can use this freedom to build products that are better in certain respects than other implementations of the specification. CORBA 1.0 did not specify a communications protocol; every ORB was allowed to employ a proprietary protocol. It was thought that this opportunity for

vendor-specific optimisations would benefit the cause of CORBA. However, the resulting proliferation of communications protocols made interoperability between different ORB implementations cumbersome. And although CORBA 2.0 still allows request broker implementations to use whatever communications protocol they prefer, it has introduced the interoperability architecture to cater for the problems of interoperability.

The interoperability architecture is built around the notion of bridges. A *bridge* sits between two request brokers and translates messages from the communications protocol used by one broker to messages of the communication protocol used by the other broker. Bridges allow communication between brokers that have no knowledge of how the other implements the CORBA standard.

In general, the communication protocols of two brokers may differ so significantly that constructing a bridge between them is very hard. The sequence of interaction of communications messages and the information they contain may differ largely. Additionally, the bridge must take into account the possibly different data encodings that the communication protocols use. To facilitate easy bridging, CORBA 2.0 has introduced a number of interoperability protocols. Figure 5 gives an overview.

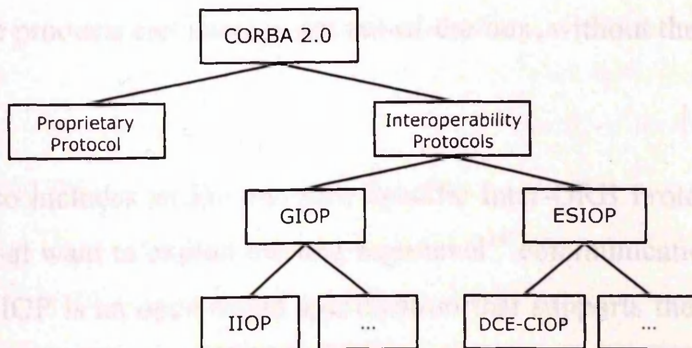

Figure 5 Communications Protocol Family

The General Inter-ORB Protocol (GIOP) is specifically built for ORB-to-ORB interactions and is designed to work directly over any connection-oriented transport protocol that meets a minimal set of assumptions. This set includes, for example, the assumptions that connections are symmetric and full duplex. Other assumptions are that the transport is reliable and that the transport indicates disorderly loss of connections. These assumptions exactly match the guarantees provided by the TCP/IP

networking protocol [Hal]. However, other transports also meet these requirements. Among others: the Systems Network Architecture (SNA) [Tan], Xerox Network Systems' Internet Transport Protocol (XNS/ITP) [Tan], Asynchronous Transfer Mode (ATM) [ATM], HyperText Transfer Protocol next Generation (HTTP-NG) [NG], and Frame Relay [Hal].

The General Inter-ORB Protocol (GIOP) specifies a standard transfer syntax and a set of message formats for communications between ORBs. We discuss these in more detail in the following sections. GIOP is not tied to any particular networking protocol, so different versions, running on different networking protocols can exist. And while these versions would not be directly interoperable, their commonality guarantees easy and efficient bridging.

The Internet Inter-ORB Protocol (IIOP) is a version of GIOP that specifies how GIOP messages are exchanged using TCP/IP connections. All request brokers must support IIOP natively or provide a half-bridge that can interpret IIOP messages. Therefore, IIOP can be seen as a canonical communications protocol to which all other protocols must be reducible. Due to this status, a lot of the commercially available ORB implementations use IIOP as their internal communications protocol [OOC, Orbix]. Therefore, these products can interoperate out-of-the-box, without the presence of any bridges.

CORBA 2.0 also includes an Environment Specific Inter-ORB Protocol (ESIOP) for environments that want to exploit existing high-level[14] communication facilities (e.g. DCE RPC). ESIOP is an open-ended specification that supports the exploitations of implementation-specific transport assumptions. While specific ESIOPs may be optimized for particular environments, all ESIOP specifications will be expected to implement half-bridges for GIOP.

Note. Because all ORB implementations have to support IIOP to some degree and because the IIOP specification is for a large part fixed by the GIOP specification, we

[14] I.e. higher-level primitives than the ones offered by a connection-oriented network protocol such as TCP/IP.

will assume in the remainder of this document the use of a GIOP-based communications protocol. Therefore, all the conclusions drawn in the next sections are only applicable to situations where the ORB implementation uses a GIOP-based protocol internally or either where several ORB implementations have to interoperate.

3.2.2 Message Formats

GIOP provides a specification of a number of communications messages, what information they contain and how interaction is achieved using them. IIOP adheres to these message formats. As we have seen, all CORBA 2.0 compliant ORBs must support IIOP, therefore, the specification of the messages and their layout enforces a certain commonality of communication protocols that facilitates easy bridging.

GIOP defines eight message types that are used by clients and servers to communicate: Request, CancelRequest, LocateRequest, Reply, LocateReply, CloseConnection, MessageError and Fragment. The first three messages can only be sent by a client, the next three can only be sent by the server; the last two can come from both sides. Only two of these messages are necessary to achieve basic remote procedure call semantics: Request and Reply. The others are control messages or messages to support certain optimisations.

All messages have the same basic layout (cf. Figure 6). Messages begin with a header of twelve bytes. The first 6 bytes contain the string 'GIOP', followed by the major and minor version numbers (v_m, v_n). This can be 1.0, 1.1 or 1.2 at the moment. The next byte (x) is a flags byte, containing information such as whether the message is little-endian or big-endian encoded, etc. Byte number eight (y) indicates the message type: 0 indicates a request message, 1 indicates a reply message, 2 indicates a CancelRequest message, etc. The last four bytes of the header contain an unsigned value that indicates the size of the message body. This value is encoded little-endian or big-endian as indicated by the flags-byte.

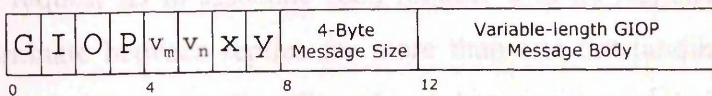

Figure 6 Basic Structure of a GIOP Message

The GIOP header is specified using a syntax called Pseudo-IDL or PIDL. This syntax allows the specification of data structures that are not regular CORBA objects but for which a language-independent type description is necessary. The following PIDL shows the GIOP header. The character array `magic` contains the letters 'GIOP', then comes the protocol version, the flags byte, message type and message length.

```
struct MessageHeader_1_2
{
   char           magic[4];
   Version        GIOP_version;
   octet          flags;
   octet          message_type;
   unsigned long  message_size;
};
```

By definition, the client is the party that opens a connection, and the server is the party that accepts the connection. To invoke an operation on an object, the client opens a connection and sends a request message. The client then waits for a reply message from the server on that connection[15]. A Request message is always sent from client to server and is used to invoke an operation or to read or write an attribute. Request messages carry all in and inout parameters that are required to invoke an operation. A Reply message is always sent from server to client, and only in response to a previous request[16]. It contains the results of an operation invocation –that is, any return value, inout parameters, and out parameters. If an operation raises an exception, the Reply message contains the exception that was raised.

3.2.2.1 Request Message

A request message also consists of a header and a body (cf. Figure 7). The entire request message is the request body of a general GIOP message. The request header contains –among other things- a request ID, an object key and an operation name. The client uses the request ID to associate each request with its response; this allows a client to discriminate between replies for more than one outstanding request when using deferred synchronous calls. The object key is extracted from the object

[15] Remember, GIOP assumes a connection-oriented network protocol.

[16] This has recently changed with the support of bi-directional communication in GIOP 1.2

reference and is used to identify the particular object in the server that the request is for. The operation field contains a string denoting the name of the operation being invoked. The operation name uniquely identifies operations within an object. This conforms to the data model as operation overloading is not allowed. The request body contains the CDR-encoded in and inout parameters in order of their definition in the IDL specification. We discuss CDR in detail in the next section.

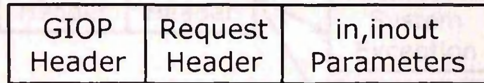

Figure 7 GIOP Request Message Layout

The request header is also specified in Pseudo-IDL. The following PIDL shows part of it:

```
//PIDL
struct RequestHeader_1_2
{
    //...
    unsigned long          request_id;
    boolean                response_expected;
    sequence<octet>        object_key;
    string                 operation;
};
```

3.2.2.2 Reply Message

Reply messages have a similar layout (cf. Figure 8). Their header contains information such as service context, request ID and reply status. The reply status indicates whether the result of the request is successful, a user exception, a system exception or a location forward message. If the reply status indicates successful completion of the operation, the message body contains the values for the out and inout parameters of the operation (again in the order of declaration in the IDL interface specification). If the reply status indicates a user or system exception, the reply body contains the information associated to the exception. If the reply status indicates a location forward message, the reply body contains a reference. Such a message indicates that the target object has moved to a new location and can be accessed through the reference contained in the message body. It is possible that this

reference has also been outdated by another relocation of the object. In that case, usage of the reference will result in another location forward message, containing a more recent indication of where the object resides.

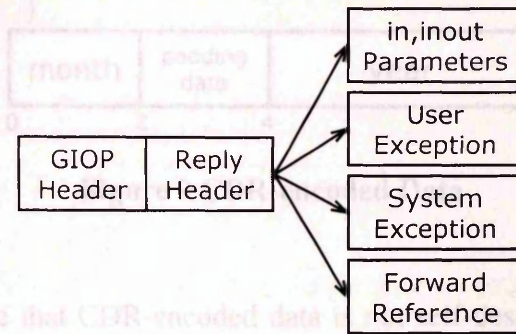

Figure 8 GIOP Reply Message Layout

3.2.3 Common Data Representation

The standard data representation of CORBA is called the Common Data representation (CDR). A standard data representation is needed, to map between the specific data representations of the different host and programming languages. CDR uses type information to encode data. It defines a standard representation for all basic types, constructed types and for object references (see next section). The exact encoding rules can be found in the specification section 15.3 [Core]. All data that flows over the network has to have a type associated with it. The GIOP header and the specific message header are specified in PIDL. The content of the message body is specified by regular IDL declarations (as part of e.g. an interface declaration, exception declaration, etc).

Figure 9 contains an example of an IDL record type declaration and how it is encoded using CDR. The record consists of two fields: month and year. Encoded, the record is 8 bytes long. The first 2 bytes contain the short value representing the month. The next 2 bytes are padding data (undefined contents) and the last 4 bytes contain the long value representing the year. The padding data has been inserted because CDR specifies that long values be aligned on a 4-byte boundary. Notice that neither the labels nor the types are encoded and that the values are encoded in the order of declaration of the fields.

```
struct Date
{
   short month;
   long  year;
}
```

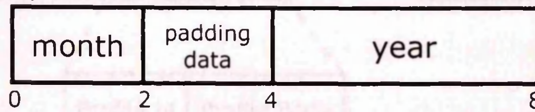

Figure 9 CDR-encoded Data

It is important to note that CDR-encoded data is not self-describing. In other words, one cannot decipher a chunk of CDR-encoded data if one does not know the type of the encoded data in advance. CDR-encoded data does not carry type information unless it is encoded by some convention together with the regular data. Some other encodings take a different approach. For example the Basic Encoding Rules (BER) used by ASN.1 [ASN] use a Tag-Length-Value (TLV) encoding, which tags each primitive data item with both its type and its length. XML-encoded data [XML] is also self-describing and may contain similar tags as BER. These encodings provide type annotations at the communication protocol level but may therefore be less efficient in both bandwidth and marshalling [ASNb].

3.2.4 Interoperable Object Reference

Object references in CORBA are opaque pieces of data that contain the information necessary to locate a server object. Every ORB vendor has the freedom to implement object references in the way he wishes. However, this means that object references cannot be passed directly between independently developed request brokers. Therefore, CDR specifies an interoperability format for object references. A reference in this format is called an Interoperable Object Reference (IOR).

The layout of an IOR is illustrated in Figure 10. It consists of a RepositoryId followed by one or more tagged profiles. A tagged profile consists of a profile id and a profile

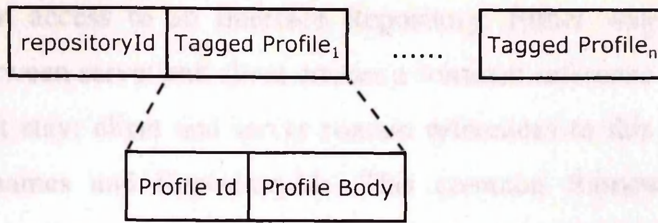

Figure 10 Structure of an IOR

body. The profile body encapsulates all the basic information needed by the protocol it supports to identify an object. An IOR is built to support more than one protocol and can contain multiple profile bodies for the same protocol. This last feature provides a hook for load balancing and fault-tolerant computing.

What protocol data is captured depends on the specific protocol. In the case of IIOP (TCP/IP), the protocol profile data contains an Internet domain name or IP address, a TCP port number and an object key. The object key is an ORB-specific identifier that is generated by the Object Adaptor of the server where the target object has been created. Instead of the address and port number of the server that implements the object, it may also contain the address of an Implementation Repository (IR) that can be consulted to locate an appropriate server. This extra level of indirection permits server processes to migrate from machine to machine without breaking existing references held by clients[17]. The object key is an ORB-specific identifier that allows the ORB to locate the target object for which it receives a request.

There also exists a null object reference. A null object reference is indicated by an empty set of profiles, and by a "Null" type ID (i.e. a string which contains only a single terminating-character).

The RepositoryId is optional. A RepositoryId is a string that identifies the most derived type of the IOR at the time the IOR was created. We discuss RepositoryIds in the next section. When a RepositoryId is contained in a reference, it gives the client an indication of the type of the target object. But the client must have a way of

[17] Another mechanism is in place for object relocation. It is embedded in the GIOP message types. The disadvantage of this mechanism is that the number of lookups may grow linearly with the number of times an object is relocated.

interpreting the RepositoryId and link it to the actual type specification. This can be achieved by giving the client access to the IDL specification at compile time or by giving the client access to an Interface Repository. Either way, just[18] passing a RepositoryId between server and client creates a common reference framework within which both must stay: client and server contain references to this framework in the form of type names and RepositoryIds. This common framework hampers the independent production and deployment of components and creates a strong cohesion between client and server. We will come back to this later (cf. Section 3.6).

3.2.5 RepositoryId

A RepositoryId is a unique string that is implicitly associated with every type declared in an IDL specification. This RepositoryId acts as a global name for the type and can be used as a key into the Interface Repository (IFR). RepositoryIds can have one of four possible formats: a format derived from IDL names, a format that uses Java class names, one that uses UUIDs and another intended for short-term use. Which format is to be used can be specified with different pragma directives in IDL.

1. The *default* IDL format:
   `IDL:`<fully qualified interface name>[`:`<version number>]
   The <fully qualified interface name> is the fully qualified programmer-defined name of the interface (i.e. name of type including namespace), prepended with a prefix (optionally specified with a prefix pragma). Prefixes have been introduced to avoid RepositoryId clashes and can be chosen to be something unique, like a trademark or a registered Internet Domain Name. In the introductory example, we have specified the prefix '`hippo.dcs.gla.ac.uk`'. The version number can be set with a pragma directive in the IDL source, but is ignored at the moment [HV p.119]. Two RepositoryIds with the same fully qualified name but a different version number are considered to be identical.
   The RepositoryId for the type Account introduced in the introductory example is:
   `IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0`.

---

[18] as opposed to passing the complete type specification

2. The *RMI Hashed* format:

RMI:<class name>:<hash code>[:<serialisation version UID>]

The RMI hashed format is used for Java RMI values mapped to IDL using the Java-to-IDL Mapping [OMGb]. It contains the class name of the original Java class and a hash code of 20 bytes, generated by the NIST Secure Hash Algorithm (SHA-1)[SHA].

3. The *DCE UUID* format:

DCE:<UUID>:<minor version number>

The DCE format contains a Universally Unique Identifier (UUID). This is equivalent to what COM calls a Globally Unique Identifier (GUID). It is a 128-bit (16 byte) number that is tool-generated, and computed according to a machine's network address and a time stamp; a combination that for practical purposes guarantees uniqueness. An example of a UUID: `B5F3E2FE-B376-11D1-BB1E-00207812E629`

4. A *local* format:

LOCAL:<user-defined type identifier>

The user-defined type identifier is an arbitrary string. The local format IDs are not intended for use outside a particular repository, and thus do not need to conform to any particular convention. They are a short-term solution, e.g. for use within a development environment.

Because RepositoryIds are unique, they are used as keys into the Interface Repository (IFR).

3.3 Server-side Request Dispatching

In Section 3.2, we have discussed the communication layer of CORBA. We have described the request messages that the client sends to the server to have an operation invoked remotely. In this section, we discuss the server-side components that interpret such request messages and dispatch them to the different servants that implement the target objects.

Figure 11 shows the different server-side components that are responsible for request dispatching. The server-side ORB core handles all the low-level networking issues

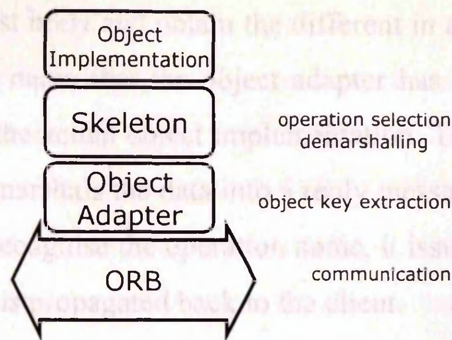

Figure 11 Server-Side Request Dispatching Components

and listens at the communication channel endpoint for GIOP messages. Because of the standardised layout of GIOP messages, it can decipher the message header and determine what sort of message it is. When a Request message is received, it extracts -again relying on the standardised layout- the different pieces of information from the Request header (cf. Section 3.2.2) and hands them over to the object adaptor, together with the Request body.

The Object Adapter is responsible for a host of tasks. It allows the ORB core to deliver operation request to an appropriate servant. In addition, it allows object implementations to use the ORB's services. The object adapter supports operations for registering new objects, activating and deactivating existing objects, incarnating and etherealising servants and accessing object information maintained by the ORB. The object adapter is also responsible for generating the IOR for a newly created object. Therefore it can interpret the object key that is embedded in the request-message header (cf. Section 3.2.2.1) and deliver the message to the skeleton of an appropriate object implementation. When an object is destroyed, it is unregistered with the object adapter. Therefore, when a message arrives for an object that has ceased to exist[19], the object key in the request message will be invalid and the object adapter will return an OBJECT_NOT_EXIST system exception.

[19] Which may happen because the lifetime of an object reference is independent from the lifetime of the object itself.

The skeleton is generated from the IDL type specification and has thus knowledge of (the signatures of) all the operations of a particular object. It can therefore decipher the CDR-encoded Request body and obtain the different in and inout parameters. If it recognises the operation name that the object adapter has forwarded to him, it can invoke the operation on the actual object implementation. The stub then waits for the results of the operation, marshals the data into a reply message and sends that back to the client. If it does not recognise the operation name, it issues a BAD_OPERATION system exception, which is propagated back to the client.

To summarise, request dispatching is done by a number of components. The ORB listens on a communication endpoint for incoming messages. The Object Adapter interprets the object key and activates a servant if necessary. The skeleton reads the operation name, unmarshals the operation parameters (based on statically embedded type information) and does the actual operation invocation.

Note, however, that the integrity of this procedure depends the ability of the server to correctly decode the CDR-encoded message body. As we have seen in Section 3.2.3, this implies that client and server must have access to the same type definitions. If client and server are out of synchronisation and the client sends a message to the server containing an operation name that the server recognises, the server would decode the message body erroneously (i.e. based on different type definitions than the ones used to encode the data). This may result in a violation of strong typing. It would then be, for example, possible for a client to provide a floating-point number to an operation expecting an integer number and this would go unnoticed by the system. Synchronisation of client and server is enforced by the development process, which demands that the IDL interface specification is distributed unchanged to all nodes ahead of further development (cf. Section 2.4).

The specification states on page 10-43 that "for interfaces, if stubs and skeletons are not actually in synch, even though the RepositoryIds report they are, the worst that can happen is that the result of an invocation is a BAD_OPERATION exception." Note that this is not true.

3.4 Client-side Operation Invocation

3.4.1 Static Operation Invocation

With the Static Invocation Interface (SII), operation invocation happens through a stub or proxy (cf. Figure 12). A stub is an implementation language entity that is automatically generated from an IDL specification. It handles the construction, sending and receiving of GIOP messages using the API of the ORB. The stub acts as a local stand-in for a remote object. It translates all incoming operation invocations into request messages and sends them to the remote object it represents. Because the stub is automatically generated for a specific interface, it can only handle

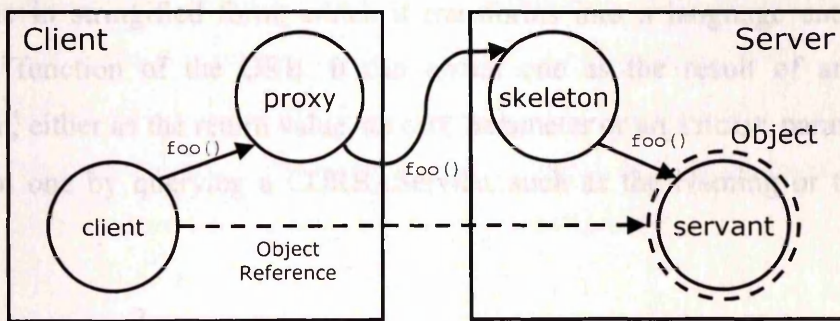

Figure 12 Communication via Stub and Skeleton

communications messages for the operations in that are specified in that interface. Therefore, it is important that the type of the proxy matches the type of the target object. In this section we investigate how and when these proxy objects are instantiated.

The specification states that a stub is created when a new object reference enters the address space of a client. The exact procedure of how proxies are instantiated depends on the language mapping of the client language. For the remainder of the discussion we will assume C++ as the client implementation language. The choice of C++ over Java[20] in this discussion is pragmatic due to the availability of excellent technical material on these matters [HV]. Although we will try to keep the language-dependent discussion to a minimum, we need to go into some detail about the C++ mapping. For a complete discussion of the C++ mapping we refer to [OMGa, HV].

[20] Java is the language we used for the implementation of the example in the introduction.

The C++ mapping defines a proxy class for each interface–usually abstract- and an object reference type. The proxy class has the same name (unqualified) as the IDL interface it represents; the object reference type has this name followed by the suffix _ptr. A client is not allowed to directly instantiate a proxy class, nor is it allowed to declare a pointer or a reference to a proxy class. All manipulations of proxies must go via instances of the automatically declared object reference type. It is always the client-side ORB run-time that creates proxies on behalf of the client when an object reference enters the client's address space.

There are three main ways in which a client can obtain an object reference. It can obtain one in stringified form, which it transforms into a language entity using a dedicated function of the ORB. It can obtain one as the result of an operation invocation, either as the return value, an out parameter or an inout parameter. Or it can obtain one by querying a CORBAService, such as the Naming or the Trading service.

1. In stringified form:

Assuming a string s containing an IOR in stringified form and the IDL specification from the introductory example compiled with an appropriate IDL-to-C++ compiler, the following code reflects how a local stub is instantiated and bound to a remote object:

```
CORBA::Object_ptr o = string_to_object(s);
```

An instance o of the object reference type for Object is declared. Object is the implicit supertype of all interface types. The reference type for object can hold references of no matter what type. The string_to_object() function takes a stringified IOR. It instantiates a stub of type Object, configures it according to the communication endpoint information contained in the IOR and returns a C++ object reference that is bound to this newly instantiated stub. In order to invoke more specific operations on the target object than the ones defined in the Object interface, the reference must be narrowed to a more specific type:

```
Account_ptr myAccount = Account::_narrow(o);
```

An object reference of type Account is declared. This type has been automatically generated from the IDL definition of the interface Account. On the generic object reference o, the static narrow function of the proxy type Account is applied. This

function checks, by using the RepositoryId, whether the type of the object reference matches the type of the stub it belongs to. We will explain this check in more detail in the section on type equivalence (3.6.1). If the check succeeds, a stub of the type Account is instantiated. From this point on, it is possible to invoke operations on the reference for the remote object as if it were a local object. If the check fails, a null reference is returned.

```
Currency balance = myAccount->balance();
//...
```

Execution of the above code results in the instantiation of two proxies to the same remote object. One of type Object and one of type Account. These proxies utilise system resources such as memory, network connections, etc. and must therefore be cleaned up after usage. Usually proxies are either automatically garbage collected or carry a reference count. The latter is the case in C++. The `CORBA::_release()` function decreases the reference count. In this example, both stubs have a reference count of 1 and are therefore cleaned up by the following code:

```
CORBA::release(o);
CORBA::release(myAccount);
```

2. *As the result of an operation*:

When a reference is obtained as the result of a remote operation invocation, be it as the return value, an `out` or `inout` parameter, its type is specified by the signature of the method in the IDL specification. The automatically generated stub-code for each method will instantiate a proxy of the type as specified in the IDL description. There is no need for an explicit check. This can be deduced from the following argument. Communication between client and server is type-safe because stub and skeleton are assumed to be in synchronisation. So, the client receives the object reference under the same type that the server sees it.

- The server has created the reference itself (i.e. the server has created the target object), in which case the object reference conforms to the type indicated in the signature.
- The server has obtained the reference from another server.
    - When it has obtained it via a mechanism that has an explicit `_narrow()` in it, type-safety is guaranteed.

- o Otherwise, it has obtained it in an unchecked manner from another server, which starts a recursive argument. This recursive argument stops because all references can either be traced to the place where they have been created, have been obtained in stringified form or have been obtained by invoking `resolve_initial_references()` which results in a reference of type Object and needs an explicit `_narrow()`.

In conclusion, the following code executes in a type-safe way without a run-time check being executed:

```
Account_ptr myAccount = manager->getAccount("David");
```

The result of the operation `getAccount()` is a reference of type Account. Behind the scenes, the stub of `manager` instantiates an Account stub and returns a reference to it.

3. *Via a CORBAService*:

Sometimes services need to be generic and can not return the exact type of what the client expects. This is often the case for standardised services such as Naming Service, Trading Service, etc. In such cases, the client must narrow the result to the type it expects. This is basically the same scenario as with a stringified reference.

```
CORBA::Object_var o = nameService->resolve(someName);
Account myAccount = Account::_narrow(o);
```

The static narrow function of the Account stub only instantiates an Account stub when nameService has indeed returned a reference to an Account object.

From discussion, we can see that no matter how an object reference enters the address space of a client, it is guaranteed to be bound to a stub of a matching type. When a reference enters the address space in a stringified form or in a generic form, it is explicitly checked by a function called `_narrow()` that is automatically generated from the type definitions. When an object reference enters the address space as the result of an operation invocation, it is guaranteed to have the correct type. This because of the assumption that stubs and skeletons are in synchronisation and the recursive argument that the reference that is passed on, is either 'home made' (it is safe because of strong typing assumption of implementation language) or obtained from another server. All references that are not 'home made', can be traced back to

initial references, which can only be obtained via a mechanism containing an explicit check.

3.4.2 Dynamic Operation Invocation

The specification states on page 7-2 that "parameters supplied to a request *may* be subject to run-time type checking upon request invocation". It is left to the different request broker implementers to decide whether or not they provide this 'feature' and if it is provided, how the check is performed.

An important class of applications employing the DII relies on the Interface Repository to obtain the type information they require to construct requests. For this kind of applications, type-checking is not very useful as the type information comes from a trusted source. However, for applications using the DII for its ability to invoke one-way or deferred synchronous requests, the absence of type-checking may prove more of a disadvantage.

The ORBacus ORB implementation [OOC] used for the example does not support type-checking of parameters in case of the DII. So, when a programmer by accident provides parameters to an operation in the wrong order, this will go undetected!

3.5 *Strong Typing in CORBA*

In the previous sections, we have discussed in detail the different mechanisms that CORBA specifies to support request invocations.

We have discussed GIOP, the communications protocol of CORBA. We have shown that the communications message for requesting an operation of an object contains the name of the operation and an ordered list of in and inout parameters, encoded in CDR. We have discussed CDR and explained that one needs to know the type of the data in advance, in order to be able to decipher CDR-encoded data.

We have also discussed the server-side request dispatching mechanisms and noted that incoming messages are interpreted according to local type declarations. No additional check is in place to assess whether the data has been encoded using the same or conforming type declarations.

At the client-side, we have discussed the different request invocation mechanisms. For static invocations, we have explained when and how stubs are instantiated and we have demonstrated that the interface of a stub always conforms to the interface of the object it represents. The checks to assert that this is indeed the case rely on RepositoryIds and the fact that they represent the identities of the different types. However, CORBA does not provide mechanical means to maintain the link between a RepositoryId and a type declaration. They match by virtue of the development process used to construct CORBA applications. Therefore, under the assumption that a CORBA application is constructed using the process outlined in the previous chapter, we can say that the SII is type-safe.

For dynamic invocations, the specification makes type-checking of parameters optional. Additionally, it does not provide information on how the check is to be done. Therefore, we can draw no general conclusions concerning strong typing in the case of the DII

3.6 Type Equivalence

Type equivalence is the criterion to which type specifications are compared to establish the correctness of computations. The criterion must preserve the mechanical correctness of a computation but yet be flexible enough to allow computations to execute as intended. A conformance check is required each time a binding is established between two denotations. Therefore, assignment between two variables also calls for a type equivalence check.

Each language makes its own trade-off between safety and flexibility and decides on its own set of rules for the equivalence check. Some languages may regard primitive types to be equivalent regardless of whether they are named or not, while treating each occurrence of a compound type declaration as a new type, regardless of whether it is named or not (e.g. C++ [Str]). Other languages solely rely on the type rules as defined in their type systems to deduce whether two types are equivalent (e.g. Napier88 [Nap]). Yet other languages allow the programmer to indicate a preference for one of the aforementioned checks (e.g. Modula-3 [Mod] with its branding mechanism). The differences in these rules result in the fact that for some languages

the types resulting from the following type declarations are equivalent, while for other languages the resulting types are not equivalent.

```
//pseudo-code
typedef Point  = record [ x : float, y : float ];
typedef Vector = record [ x : float, y : float ];
```

This means that in some languages execution of the function application in the following pseudo-code fragment, will result in a type error, while in others it will execute successfully.

```
//pseudo-code
void f(x : Vector)
a : Point;
...
f(a);   //may or may not result in an error
```

3.6.1 Type Equivalence in CORBA

Every type is identified by its RepositoryId. Therefore, in CORBA, two types are considered to be equivalent if their RepositoryIds are identical. The CORBA type system allows subtyping between interfaces. More types may be considered to be equivalent based on the explicit inheritance relationships defined in an IDL specification.

The type equivalence check is performed by a method called _narrow, which is automatically generated for each interface. _narrow takes an object reference and tries to deduce whether the interface associated with the RepositoryId conforms to the interface that this particular _narrow function is associated with. If the object reference does not contain a RepositoryId or if the client cannot deduce conformance, the target object of the object reference is contacted. For this purpose, a method is available in the interface of Object (the root of all interfaces): is_a. This method takes a RepositoryId and returns true if the object on which it is requested conforms to the interface associated to the RepositoryId. If conformance is assessed, a stub is instantiated for the remote object.

The specification requires that the equivalence test cannot fail before the server has been contacted. In fact, ORB implementations that solely rely on the is_a-method for the implementation of the different _narrow-methods are compliant. In such implementations, binding will always result in contacting the server. The ORBacus ORB implementation [OOC] used to implement the examples (cf. appendices) for a large part uses this approach and contacts the server for most bindings.

Contacting the server to determine type equivalence allows for a limited amount of server-side evolution. A server-object can be replaced with another server-object that has a more specialised interface without the necessity to redeploy all its clients. In such cases, a client can access a server for which it has no static knowledge of the type. However, this flexibility is traded for performance as every binding of such a kind results in a remote operation invocation.

In CORBA, binding a variable to a remote object results in two equivalence checks. The first check is the check performed by the narrow function as discussed above. When it succeeds, a stub is instantiated and a second check is performed. This check is a check by the implementation language, to assert that the type of the stub conforms to the type of the variable it is assigned to. As one of the goals of CORBA (or distributed middleware in general) is orthogonal distribution, these two checks should blend together to provide the application programmer with an intelligible check that fits in a logical way with other features of the implementation language. However, this is not the approach currently taken by CORBA. The client cannot decide on its own equivalence check, as it has to contact the server in certain cases (cf supra). It would, for example, be hard to construct a language mapping that conforms to the standard but that employs a Napier88 like type equivalence scheme.

The fact that a _narrow may contact the server possibly poses a performance problem for certain applications, e.g. applications that cannot depend on the unpredictable response times of the network. Therefore, it is rumoured [New] that a future version of the specification will contain an *unchecked narrow*. Needless to say that strong typing is not preserved when this narrow is employed.

3.7 *Interface Evolution*

Consider the bank account application of Section 2.6 and imagine the changes induced by the introduction of the Euro. As long as the currency was Belgian Franks, deposits and withdrawals could only happen by integer amounts. With the advent of the Euro, this is no longer the case. Therefore, we have to model such amounts with floating point numbers rather than with integer numbers. Luckily, we had introduced a type Currency. So the only change we need to make to the IDL specification is the alteration of the definition of the type `Currency` in the module `BankApplication` from `unsigned long` to `float`. The rest of the specification can remain untouched.

The IDL specification has to be recompiled to generate type declarations, stubs and skeletons that reflect the changes we have made. The interface implementation of Account and all its clients have to be adapted to cope with Euros. And the server-program has to be restarted. When all this is done, application usage can be resumed.

But the Account interface may be used by a large number of clients, and it may be difficult to track them all. So what would happen if one client would remain unchanged and obtain a reference to a 'new' `Account` object?

Assuming that the default RepositoryId format is being used, the reference would contain a RepositoryId identical to the one that it contained before, because the interface name has not been changed (see Section 3.2.5). Therefore, the check performed by the `Account::_narrow()` function in the client stub would still succeed and the client would successfully bind to the object. But the narrow-function has been generated with respect to the old interface, where the type Currency was still declared as `unsigned long` rather than `float`. Any invocations made by the client on the stub may result in requests containing data encoded as unsigned long. Unfortunately, the size of a CDR-encoded `unsigned long` is the same as the size of a CDR-encode `float`. Therefore, as we have seen in Section 3.3, the server cannot discriminate between messages from 'new' clients and messages from unmodified clients. When the server receives a message from an old client, he interprets the values, which have been inserted as `unsigned longs`, as `floats`.

The resulting arbitrary values may invalidate the state of the server and any manipulations of this value are a violation of strong typing according to the client's type declarations.

The underlying reason for the previous unsound behaviour is the fact that the RepositoryId has not been changed, while the interface has. In the whole of the CORBA specification, there is the silent assumption that a RepositoryId represents the identity of an interface specification. But CORBA does not offer any mechanical means to maintain this implicit link between a RepositoryId and the type structure from which it is generated. This is fine in a static world, but when interfaces evolve, the same RepositoryId may be reused for different, incompatible interfaces over time, resulting in the aforementioned problems.

The only way to overcome problems such as these with a client that is out of synchronisation, is to change the interface specification in such a way that the RepositoryId changes. Simply adding a version number to the interface (with the version pragma) does *not* solve the problem, as the version numbers are not taken into account for RepositoryId comparison [HV p.119].

However, with the default RepositoryId format, changing the RepositoryId that is generated from an interface is not straightforward. It is after all generated from a prefix (e.g. something unique such as a trademark or an Internet domain name) and a logical interface name. These two things are not very likely to change when small adaptations are made to an interface. Therefore, it may be better to use another RepositoryId format. Using the RMI hashed format, the RepositoryId automatically changes when the interface changes, be it not in all cases[21] and would normally only be used in conjunction with Java. It may be better to utilise the UUID format and generate a new UUID every time a change to the interface is made. The fact that there is no obvious logical connection between a UUID and an interface means that changing the UUID after every change of interface is more intelligible for the

---

[21] Due to the fact that a fixed length hash value is used, two different type structures can be mapped onto the same value.

programmer than making changes to the specification so that the default RepositoryId changes.

It is interesting to note that some changes to an interface are harmless. Just adding new operations, for example, cannot result in a breach of the server-side encapsulation. The stub of the unmodified client can only send a subset of the messages that the updated skeleton can understand. The normal practice (see discussion in appendices) in such cases should be to introduce a new interface that inherits from the old. When this approach is followed, a client with no static knowledge of this new interface, will try to bind to such an new object by utilising the `_is_a()` function of the (generic) stub that it has instantiated. This will send the RepositoryId of the most derived interface that the client has knowledge of to the server. The server can assess that its own interface is in fact a subtype of the interface associated to the RepositoryId it receives. Therefore, the binding can succeed and the client can instantiate the interface-specific stub. Because of the fact that the interface of this stub is a strict subset of the interface of the target object, it will generate only request messages that the server object's skeleton can handle. The advantage of just adding new operations to an interface, rather than generating a new type, is that even without redeployment, no performance penalty has to be made for bindings.

A heated debate on whether it is allowed to add new operations to an interface, without redeploying the clients, can be found in the appendices. Several CORBA gurus fiercely disagree on this question. This is an indication of the fact that the specification is not always clear and that deep insight into these matters is not widespread.

The wider implications of all this is that in CORBA client and server are rather tightly coupled. Any change to the interface of a server implies redeployment of all its clients. Failure to do so may not result in compile-time or run-time errors, but in hard to find logical errors. Additionally, client and server are created with respect to a common reference framework of type declarations. Both sides must theoretically have access to the complete specifications of all types that are used in a system. It is hard to

maintain such a way of working in the face of global networks such as the Internet, where autonomy of client and server is an important property.

3.7.1 A very short note on DCOM

In DCOM interfaces also have a run-time identity (equivalent to a RepositoryId). There are no formats to choose from. All interfaces are identified by a GUID. The GUID uniquely identifies that interface over time and space. When an interface is changed, DCOM regards it as a different interface and therefore, a different GUID must be assigned to it. This is very similar to what happens in CORBA. The difference between CORBA and DCOM is that DCOM assumes that a tool generates its type definitions. When a program is developed using e.g. J++, the interfaces are written in Java and then exported to DCOM. A tool automatically converts the Java interface definitions into MS IDL, generates a GUID and inserts it textually into the specification. It then utilises the Microsoft-provided IDL compiler to generate the stubs and skeletons (which are called proxy and stub respectively) from this specification.

One expects that with tool support, a change in an interface automatically results in the generation of a new GUID (something that is necessary, as we have seen above, to guard correct execution semantics). However, in J++, exporting an interface to DCOM happens in two phases. The programmer must first indicate that he wants a certain class to be available in DCOM. The environment then pastes a GUID textually in a comment field of the source code. From then on, compilation also generates the stub and skeletons and registers the class in the registry under the GUID specified in that comment field. With this way of working, however, recompilation of a changed interface does not necessarily result in the introduction of a new GUID. For that to happen, the programmer has to delete and regenerate the comment-field. Therefore, it can be concluded that DCOM has the same problems as CORBA with interface evolution.

4 Conclusions

Middleware, such as CORBA, is widely employed to integrate different applications into a heterogeneous distributed application. An important aspect of middleware is that it transparently extends programming languages to enable access to objects in different address spaces. This thesis presents an investigation into the mechanisms specified by CORBA for such an extension and assesses whether these mechanisms, when applied to a strongly typed language, still guarantee strongly typed execution semantics.

We have discussed the communications layer of CORBA. In particular we have discussed the General Inter-ORB Protocol, which is the protocol specified for the most general usage scenario. GIOP overcomes the heterogeneity of data representations of different platforms and languages by adopting the Common Data Representation (CDR). Transforming data in and out of this encoding happens on basis of the type of that data. Two parties can only successfully exchange data when they encode/decode it with respect to matching type declarations.

We have discussed that in CORBA all type declarations are global and are supposed to be uniquely identified by a RepositoryId. We have noted that CORBA has no mechanism in place to mechanically enforce this uniqueness over time and space.

We have discussed the server-side request dispatching mechanism. Several components work together to dispatch a request to the correct object implementation: the server-side request broker, the object adapter(s) and the skeleton. The main observation that we made in this context is that operations are only identified by their name (and the object they belong to). The server-side components cannot determine from an incoming GIOP message whether the parameters supplied to the operation are valid. Therefore, the interface of the object can potentially be bypassed or an operation may be invoked with an undefined set of parameters.

We have discussed the two different request invocation mechanisms provided by CORBA: the Static Invocation Interface (SII) and the Dynamic Invocation Interface (DII).

* The SII relies on stubs that have been generated at compile-time from the IDL type declarations. When a remote object is bound to a local variable, a check is sometimes performed. We have outlined in which cases this check is performed and in which cases it is not. The check itself is based on the repositoryIds of the type of the local variable and the type of the remote object. The specification states that the check should not fail without contacting the server. We have seen that if there is no exact match between the repositoryIds, most ORB implementations resort to contacting the server. Therefore, the use of polymorphism is usually penalised by an additional network roundtrip. Because the check is based on repositoryIds, it critically depends on the integrity of the links between the repositoryIds and the type declarations at both client and server-side. This link is not mechanically enforced by CORBA, but it is implicitly guarded by the development process of CORBA applications.

  We have shown how a check based on RepositoryIds restricts the possibilities for interface evolution and how it creates room for easy to introduce errors that are possibly hard to detect.

  Based on all of the above observations, we have concluded that using the SII, CORBA preserves the strong typing of strongly typed languages when the application development procedure is correctly followed.

* Sometimes, it is not possible to have compile-time knowledge of the types of all the objects that an application has to handle. This is, for example, the case for gateways, object browsers, etc. The DII allows dynamic construction of requests by treating requests as objects that are created, populated, invoked and destroyed at run-time. The specification states that type-checking of the parameters of a dynamically created request is optional. Additionally, neither an API nor a type-checking procedure is specified. Therefore, whether strong typing is preserved when using the DII depends on the ORB implementation.

We have also discussed the notion of *type equivalence* in CORBA. We have noted that the central definition of CORBA type equivalence violates the programmer intelligibility of the binding of a programming language variable to a remote object.

The main contribution of this work is the provision of an in-depth discussion of the main components that guard type soundness in CORBA from a programmer's viewpoint, but at a level of abstraction that transcends a mere how-to discussion. This information is hard to come by, but essential in order to successfully utilise complex frameworks such as CORBA. The vast complexity of the specification prohibits it from serving as a guideline. Yet the how-to oriented textbooks provide only superficial understanding and require a high degree of re-learning when switching to a different language or even a different ORB implementation.

Discussing CORBA by identifying common programming language concepts such as strong typing, type equivalence, naming and binding has allowed us to reach a high level of abstraction without losing technical accurateness. It can be expected that this approach can be taken further to shed more light on CORBA, or for that matter any complex programming framework, such as DCOM, EJB, etc.

References

[ASN] ITU-T Recommendation X.690 (1997) | ISO/IEC 8825-1:1998, *Information Technology - ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*

[ASNb] *ASN.1 Homepage*; http://www-sop.inria.fr/rodeo/personnel/hoschka/asn1.html

[ATM] The ATM Protocol - http://www.atmforum.com/atmforum/specs/specs.html

[Box] D. Box – *Essential Com*, Addison-Wesley 1998; ISBN 0-201-63446-5.

[BS] R. Bastide, O. Sy – *Towards Components that Plug AND Play*. ECOOP'2000 Workshop on Object Interoperability, Nice, France, June 2000.

[BW] M. Büchi & W. Weck – *Java Needs Compound Types*, Technical Report nr. 182, Turku Centre for Computer Science.

[CBC] Connor, R. C. H., Brown, A. B., Cutts, Q. I., Dearle, A., Morrison, R. & Rosenberg, J. – *Type Equivalence Checking in Persistent Object Systems*, in Implementing Persistent Object Bases, Principles and Practice, A. Dearle, G. M. Shaw and S. B. Zdonik (ed.), Morgan Kaufmann, Proc. 4th International Workshop on Persistent Object Systems, Martha's Vineyard, USA pp 151-164 (1990).

[Con] R. Connor – *Types and Polymorphism in Persistent Programming* Systems, Ph.D. Thesis, University of St Andrews (1990).

[Core] OMG – *The Common Object Request Broker: Architecture and Specification*. Revision 2.4, October 2000. ftp://www.omg.org/pub/docs/formal/00-10-01.pdf.

[CW85] L. Cardelli and P. Wegner - *On Understanding Types, Data Abstraction and Polymorphism*; ACM Computing Surveys 17, 4 (December 1985) pp. 471 – 523

[Em96] W. Emmerich – *Genericity and Interoperability in CORBA*(lecture), http://www.cs.ucl.ac.uk/staff/W.Emmerich/lectures/DS96-97/

[Em00] W. Emmerich – *Engineering Distributed Objects*, John Wiley & Sons 2000; ISBN 0-471-98657-7.

[Hal] F. Halsall – *Data Communications, Computer Networks and Open Systems*, Addison-Wesley; ISBN: 0-20-142293-X

[HV99] M. Henning & S. Vinoski – *Advanced CORBA Programming with C++*, *Addison*-Wesley Professional Computing Series 1999; ISBN 0-201-37927-9.

[Mod] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow & G. Nelson - *(Modula-3) Language definition*, http://www.luca.demon.co.uk/Bibliography.html

[MS] Microsoft – *The COM Specification*;

http://www.microsoft.com/com/resources/comdocs.asp

[Nap] R. Morrison, A. Brown, R. Connor, Q. Cutts, A. Dearle, G. Kirby & D. Munro - *The Napier88 Reference Manual (Release 2.0)*, University of St Andrews Technical Report CS/94/8 (1994).

[New] Internet Newsgroup, comp.object.corba

[NG] The HTTP-NG Prtotocol - http://www.w3.org/Protocols/HTTP-NG/

[OMA] OMG – *Object Management Architecture*. January 1997.

[OMGa] OMG – *C++ Language Mapping*. June 1999. ftp://www.omg.org/pub/docs/formal/99-07-45.pdf.

[OMGb] OMG – *IDL/Java Language Mapping*. June 1999. ftp://www.omg.org/pub/docs/formal/99-07-53.pdf.

[OOC] Object Oriented Concepts – *ORBacus for Java*. http://www.ooc.com

[Orbix] Iona – Orbix 2000; http://www.iona.com

[Pri] J. Pritchard – *COM and CORBA Side by Side: Architectures, Strategies, and Implementations*; ISBN 0-201-37945-7

[Rit] F. Rittinger - *Sicherheit in Verteilte Systemen - CORBA Sicherheitsdienst* (talk)

[Sch90] D. A. Schmidt – *The Structure of Typed Programming Languages*, MIT Press; ISBN 0-262-19349-3.

[SHA] Secure Hash Algorithm - http://csrc.nist.gov/cryptval/shs.html

[Str] B. Stroustrup -*The C++ Programming Language, Special Edition*; Longman Higher Education; ISBN: 0-20-170073-5

[Szyp] C. Szyperski – *Component Software: Beyond Object-Oriented Programming*, ACM Press, Addison-Wesley 1997; ISBN 0-201-17888-5.

[Vin98] S. Vinoski – *New Features for CORBA 3.0*, Communications of the ACM, Vol. 41, No. 10, October 1998.

[XML] W3C – *XML Specification*; http://www.w3.org/XML/

Appendix I

Source Code for Introductory Example

This appendix contains the full source of the introductory example. The following files are included:

- `Account.java`
- `AccountOperations.java`
- `AccountHelper.java`
- `AccountHolder.java`
- `AccountPOA.java`
- `_AccountStub.java`
- `Account_impl.java`

- `AccountManager.java`
- `AccountManagerOperations.java`
- `AccountManagerHelper.java`
- `AccountManagerHolder.java`
- `AccountManagerPOA.java`
- `_AccountManagerStub.java`
- `AccountManager_impl.java`

- `CurrencyHelper.java`

- `Client.java`
- `Server.java`

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0
//
/***/

public interface Account extends AccountOperations,
                                 org.omg.CORBA.Object,
                                 org.omg.CORBA.portable.IDLEntity
{
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0
//
/***/

public interface AccountOperations
{
    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/accountHolder:1.0
    //
    /***/

    public String
    accountHolder();

    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/balance:1.0
    //
    /***/

    public int
    balance();

    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/deposit:1.0
    //
    /***/

    public void
    deposit(int amount);

    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/withdraw:1.0
    //
    /***/

    public int
    withdraw(int amount);
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0
//
final public class AccountHelper
{
    public static void
    insert(org.omg.CORBA.Any any, Account val)
    {
        org.omg.CORBA.portable.OutputStream out = any.create_output_stream();
        write(out, val);
        any.read_value(out.create_input_stream(), type());
    }

    public static Account
    extract(org.omg.CORBA.Any any)
    {
        if(any.type().equivalent(type()))
            return read(any.create_input_stream());
        else
            throw new org.omg.CORBA.BAD_OPERATION();
    }

    private static org.omg.CORBA.TypeCode typeCode_;

    public static org.omg.CORBA.TypeCode
    type()
    {
        if(typeCode_ == null)
        {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
            typeCode_ = orb.create_interface_tc(id(), "Account");
        }

        return typeCode_;
    }

    public static String
    id()
    {
        return "IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0";
    }

    public static Account
    read(org.omg.CORBA.portable.InputStream in)
    {
        org.omg.CORBA.Object _ob_v = in.read_Object();
        if(_ob_v == null)
            return null;

        try
        {
            return (Account)_ob_v;
        }
        catch(ClassCastException ex)
```

```
        {
        }

        org.omg.CORBA.portable.ObjectImpl _ob_impl;
        _ob_impl = (org.omg.CORBA.portable.ObjectImpl)_ob_v;
        _AccountStub _ob_stub = new _AccountStub();
        _ob_stub._set_delegate(_ob_impl._get_delegate());
        return _ob_stub;
    }

    public static void
    write(org.omg.CORBA.portable.OutputStream out, Account val)
    {
        out.write_Object(val);
    }

    public static Account
    narrow(org.omg.CORBA.Object val)
    {
        if(val != null)
        {
            try
            {
                return (Account)val;
            }
            catch(ClassCastException ex)
            {
            }

            if(val._is_a(id()))
            {
                org.omg.CORBA.portable.ObjectImpl _ob_impl;
                _AccountStub _ob_stub = new _AccountStub();
                _ob_impl = (org.omg.CORBA.portable.ObjectImpl)val;
                _ob_stub._set_delegate(_ob_impl._get_delegate());
                return _ob_stub;
            }

            throw new org.omg.CORBA.BAD_PARAM();
        }

        return null;
    }
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0
//
final public class AccountHolder implements org.omg.CORBA.portable.Streamable
{
    public Account value;

    public
    AccountHolder()
    {
    }

    public
    AccountHolder(Account initial)
    {
        value = initial;
    }

    public void
    _read(org.omg.CORBA.portable.InputStream in)
    {
        value = AccountHelper.read(in);
    }

    public void
    _write(org.omg.CORBA.portable.OutputStream out)
    {
        AccountHelper.write(out, value);
    }

    public org.omg.CORBA.TypeCode
    _type()
    {
        return AccountHelper.type();
    }
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0
//
public abstract class AccountPOA
    extends org.omg.PortableServer.Servant
    implements org.omg.CORBA.portable.InvokeHandler,
               AccountOperations
{
    static final String[] _ob_ids_ =
    {
        "IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0",
    };

    public Account
    _this()
    {
        return AccountHelper.narrow(super._this_object());
    }

    public Account
    _this(org.omg.CORBA.ORB orb)
    {
        return AccountHelper.narrow(super._this_object(orb));
    }

    public String[]
    _all_interfaces(org.omg.PortableServer.POA poa, byte[] objectId)
    {
        return _ob_ids_;
    }

    public org.omg.CORBA.portable.OutputStream
    _invoke(String opName,
            org.omg.CORBA.portable.InputStream in,
            org.omg.CORBA.portable.ResponseHandler handler)
    {
        final String[] _ob_names =
        {
            "_get_accountHolder",
            "_get_balance",
            "deposit",
            "withdraw"
        };

        int _ob_left = 0;
        int _ob_right = _ob_names.length;
        int _ob_index = -1;

        while(_ob_left < _ob_right)
        {
            int _ob_m = (_ob_left + _ob_right) / 2;
            int _ob_res = _ob_names[_ob_m].compareTo(opName);
            if(_ob_res == 0)
            {
```

```
                _ob_index = _ob_m;
                break;
            }
            else if(_ob_res > 0)
                _ob_right = _ob_m;
            else
                _ob_left = _ob_m + 1;
        }

        switch(_ob_index)
        {
        case 0: // _get_accountHolder
            return _OB_att_get_accountHolder(in, handler);

        case 1: // _get_balance
            return _OB_att_get_balance(in, handler);

        case 2: // deposit
            return _OB_op_deposit(in, handler);

        case 3: // withdraw
            return _OB_op_withdraw(in, handler);
        }

        throw new org.omg.CORBA.BAD_OPERATION();
    }

    private org.omg.CORBA.portable.OutputStream
    _OB_att_get_accountHolder(org.omg.CORBA.portable.InputStream in,
                              org.omg.CORBA.portable.ResponseHandler handler)
    {
        String _ob_r = accountHolder();
        org.omg.CORBA.portable.OutputStream out = handler.createReply();
        out.write_string(_ob_r);
        return out;
    }

    private org.omg.CORBA.portable.OutputStream
    _OB_att_get_balance(org.omg.CORBA.portable.InputStream in,
                        org.omg.CORBA.portable.ResponseHandler handler)
    {
        int _ob_r = balance();
        org.omg.CORBA.portable.OutputStream out = handler.createReply();
        CurrencyHelper.write(out, _ob_r);
        return out;
    }

    private org.omg.CORBA.portable.OutputStream
    _OB_op_deposit(org.omg.CORBA.portable.InputStream in,
                   org.omg.CORBA.portable.ResponseHandler handler)
    {
        org.omg.CORBA.portable.OutputStream out = null;
        int _ob_a0 = CurrencyHelper.read(in);
        deposit(_ob_a0);
        out = handler.createReply();
        return out;
    }

    private org.omg.CORBA.portable.OutputStream
    _OB_op_withdraw(org.omg.CORBA.portable.InputStream in,
                    org.omg.CORBA.portable.ResponseHandler handler)
    {
        org.omg.CORBA.portable.OutputStream out = null;
        int _ob_a0 = CurrencyHelper.read(in);
        int _ob_r = withdraw(_ob_a0);
        out = handler.createReply();
        CurrencyHelper.write(out, _ob_r);
        return out;
    }
}
```

```
// ***********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// ***********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0
//
public class _AccountStub extends org.omg.CORBA.portable.ObjectImpl
                          implements Account
{
    private static final String[] _ob_ids_ =
    {
        "IDL:hippo.dcs.gla.ac.uk/BankApplication/Account:1.0",
    };

    public String[]
    _ids()
    {
        return _ob_ids_;
    }

    final public static java.lang.Class _ob_opsClass = AccountOperations.class;

    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/accountHolder:1.0
    //
    public String
    accountHolder()
    {
        while(true)
        {
            if(!this._is_local())
            {
                org.omg.CORBA.portable.OutputStream out = null;
                org.omg.CORBA.portable.InputStream in = null;
                try
                {
                    out = _request("_get_accountHolder", true);
                    in = _invoke(out);
                    String _ob_r = in.read_string();
                    return _ob_r;
                }
                catch(org.omg.CORBA.portable.RemarshalException _ob_ex)
                {
                    continue;
                }
                catch(org.omg.CORBA.portable.ApplicationException _ob_aex)
                {
                    final String _ob_id = _ob_aex.getId();
                    throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception
                        : " + _ob_id);
                }
                finally
                {
                    _releaseReply(in);
                }
            }
            else
```

```
        {
            org.omg.CORBA.portable.ServantObject _ob_so = _servant_preinvoke    ↙
                ("accountHolder", _ob_opsClass);
            if(_ob_so == null)
                continue;
            AccountOperations _ob_self = (AccountOperations)_ob_so.servant;
            try
            {
                return _ob_self.accountHolder();
            }
            finally
            {
                _servant_postinvoke(_ob_so);
            }
        }
    }
}

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/balance:1.0
//
public int
balance()
{
    while(true)
    {
        if(!this._is_local())
        {
            org.omg.CORBA.portable.OutputStream out = null;
            org.omg.CORBA.portable.InputStream in = null;
            try
            {
                out = _request("_get_balance", true);
                in = _invoke(out);
                int _ob_r = CurrencyHelper.read(in);
                return _ob_r;
            }
            catch(org.omg.CORBA.portable.RemarshalException _ob_ex)
            {
                continue;
            }
            catch(org.omg.CORBA.portable.ApplicationException _ob_aex)
            {
                final String _ob_id = _ob_aex.getId();
                throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception    ↙
                    : " + _ob_id);
            }
            finally
            {
                _releaseReply(in);
            }
        }
        else
        {
            org.omg.CORBA.portable.ServantObject _ob_so = _servant_preinvoke    ↙
                ("balance", _ob_opsClass);
            if(_ob_so == null)
                continue;
            AccountOperations _ob_self = (AccountOperations)_ob_so.servant;
            try
            {
                return _ob_self.balance();
            }
            finally
            {
                _servant_postinvoke(_ob_so);
            }
        }
    }
}
```

```
//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/deposit:1.0
//
public void
deposit(int _ob_a0)
{
    while(true)
    {
        if(!this._is_local())
        {
            org.omg.CORBA.portable.OutputStream out = null;
            org.omg.CORBA.portable.InputStream in = null;
            try
            {
                out = _request("deposit", true);
                CurrencyHelper.write(out, _ob_a0);
                in = _invoke(out);
                return;
            }
            catch(org.omg.CORBA.portable.RemarshalException _ob_ex)
            {
                continue;
            }
            catch(org.omg.CORBA.portable.ApplicationException _ob_aex)
            {
                final String _ob_id = _ob_aex.getId();
                in = _ob_aex.getInputStream();

                throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception
                    : " + _ob_id);
            }
            finally
            {
                _releaseReply(in);
            }
        }
        else
        {
            org.omg.CORBA.portable.ServantObject _ob_so = _servant_preinvoke
                ("deposit", _ob_opsClass);
            if(_ob_so == null)
                continue;
            AccountOperations _ob_self = (AccountOperations)_ob_so.servant;
            try
            {
                _ob_self.deposit(_ob_a0);
                return;
            }
            finally
            {
                _servant_postinvoke(_ob_so);
            }
        }
    }
}

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Account/withdraw:1.0
//
public int
withdraw(int _ob_a0)
{
    while(true)
    {
        if(!this._is_local())
        {
            org.omg.CORBA.portable.OutputStream out = null;
            org.omg.CORBA.portable.InputStream in = null;
            try
```

```
                {
                    out = _request("withdraw", true);
                    CurrencyHelper.write(out, _ob_a0);
                    in = _invoke(out);
                    int _ob_r = CurrencyHelper.read(in);
                    return _ob_r;
                }
                catch(org.omg.CORBA.portable.RemarshalException _ob_ex)
                {
                    continue;
                }
                catch(org.omg.CORBA.portable.ApplicationException _ob_aex)
                {
                    final String _ob_id = _ob_aex.getId();
                    in = _ob_aex.getInputStream();

                    throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception
                        : " + _ob_id);
                }
                finally
                {
                    _releaseReply(in);
                }
            }
            else
            {
                org.omg.CORBA.portable.ServantObject _ob_so = _servant_preinvoke
                    ("withdraw", _ob_opsClass);
                if(_ob_so == null)
                    continue;
                AccountOperations _ob_self = (AccountOperations)_ob_so.servant;
                try
                {
                    return _ob_self.withdraw(_ob_a0);
                }
                finally
                {
                    _servant_postinvoke(_ob_so);
                }
            }
        }
    }
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0
//
/***/

public interface AccountManager extends AccountManagerOperations,
                                        org.omg.CORBA.Object,
                                        org.omg.CORBA.portable.IDLEntity
{
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0
//
/***/

public interface AccountManagerOperations
{
    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager/open:1.0
    //
    /***/

    public Account
    open(String name);
    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager/close:1.0
    //
    /***/

    public void
    close(String name);

    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager/get:1.0
    //
    /***/

    public Account
    get(String name);
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0
//
final public class AccountManagerHelper
{
    public static void
    insert(org.omg.CORBA.Any any, AccountManager val)
    {
        org.omg.CORBA.portable.OutputStream out = any.create_output_stream();
        write(out, val);
        any.read_value(out.create_input_stream(), type());
    }

    public static AccountManager
    extract(org.omg.CORBA.Any any)
    {
        if(any.type().equivalent(type()))
            return read(any.create_input_stream());
        else
            throw new org.omg.CORBA.BAD_OPERATION();
    }

    private static org.omg.CORBA.TypeCode typeCode_;

    public static org.omg.CORBA.TypeCode
    type()
    {
        if(typeCode_ == null)
        {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
            typeCode_ = orb.create_interface_tc(id(), "AccountManager");
        }

        return typeCode_;
    }

    public static String
    id()
    {
        return "IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0";
    }

    public static AccountManager
    read(org.omg.CORBA.portable.InputStream in)
    {
        org.omg.CORBA.Object _ob_v = in.read_Object();
        if(_ob_v == null)
            return null;

        try
        {
            return (AccountManager)_ob_v;
        }
        catch(ClassCastException ex)
```

```
        {
        }

        org.omg.CORBA.portable.ObjectImpl _ob_impl;
        _ob_impl = (org.omg.CORBA.portable.ObjectImpl)_ob_v;
        _AccountManagerStub _ob_stub = new _AccountManagerStub();
        _ob_stub._set_delegate(_ob_impl._get_delegate());
        return _ob_stub;
    }

    public static void
    write(org.omg.CORBA.portable.OutputStream out, AccountManager val)
    {
        out.write_Object(val);
    }

    public static AccountManager
    narrow(org.omg.CORBA.Object val)
    {
        if(val != null)
        {
            try
            {
                return (AccountManager)val;
            }
            catch(ClassCastException ex)
            {
            }

            if(val._is_a(id()))
            {
                org.omg.CORBA.portable.ObjectImpl _ob_impl;
                _AccountManagerStub _ob_stub = new _AccountManagerStub();
                _ob_impl = (org.omg.CORBA.portable.ObjectImpl)val;
                _ob_stub._set_delegate(_ob_impl._get_delegate());
                return _ob_stub;
            }

            throw new org.omg.CORBA.BAD_PARAM();
        }

        return null;
    }
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0
//
final public class AccountManagerHolder implements org.omg.CORBA.portable.Streamable
{
    public AccountManager value;

    public
    AccountManagerHolder()
    {
    }

    public
    AccountManagerHolder(AccountManager initial)
    {
        value = initial;
    }

    public void
    _read(org.omg.CORBA.portable.InputStream in)
    {
        value = AccountManagerHelper.read(in);
    }

    public void
    _write(org.omg.CORBA.portable.OutputStream out)
    {
        AccountManagerHelper.write(out, value);
    }

    public org.omg.CORBA.TypeCode
    _type()
    {
        return AccountManagerHelper.type();
    }
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0
//
public abstract class AccountManagerPOA
    extends org.omg.PortableServer.Servant
    implements org.omg.CORBA.portable.InvokeHandler,
               AccountManagerOperations
{
    static final String[] _ob_ids_ =
    {
        "IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0",
    };

    public AccountManager
    _this()
    {
        return AccountManagerHelper.narrow(super._this_object());
    }

    public AccountManager
    _this(org.omg.CORBA.ORB orb)
    {
        return AccountManagerHelper.narrow(super._this_object(orb));
    }

    public String[]
    _all_interfaces(org.omg.PortableServer.POA poa, byte[] objectId)
    {
        return _ob_ids_;
    }

    public org.omg.CORBA.portable.OutputStream
    _invoke(String opName,
            org.omg.CORBA.portable.InputStream in,
            org.omg.CORBA.portable.ResponseHandler handler)
    {
        final String[] _ob_names =
        {
            "close",
            "get",
            "open"
        };

        int _ob_left = 0;
        int _ob_right = _ob_names.length;
        int _ob_index = -1;

        while(_ob_left < _ob_right)
        {
            int _ob_m = (_ob_left + _ob_right) / 2;
            int _ob_res = _ob_names[_ob_m].compareTo(opName);
            if(_ob_res == 0)
            {
                _ob_index = _ob_m;
```

```
                break;
            }
            else if(_ob_res > 0)
                _ob_right = _ob_m;
            else
                _ob_left = _ob_m + 1;
        }

        switch(_ob_index)
        {
        case 0: // close
            return _OB_op_close(in, handler);

        case 1: // get
            return _OB_op_get(in, handler);

        case 2: // open
            return _OB_op_open(in, handler);
        }

        throw new org.omg.CORBA.BAD_OPERATION();
    }

    private org.omg.CORBA.portable.OutputStream
    _OB_op_close(org.omg.CORBA.portable.InputStream in,
                 org.omg.CORBA.portable.ResponseHandler handler)
    {
        org.omg.CORBA.portable.OutputStream out = null;
        String _ob_a0 = in.read_string();
        close(_ob_a0);
        out = handler.createReply();
        return out;
    }

    private org.omg.CORBA.portable.OutputStream
    _OB_op_get(org.omg.CORBA.portable.InputStream in,
               org.omg.CORBA.portable.ResponseHandler handler)
    {
        org.omg.CORBA.portable.OutputStream out = null;
        String _ob_a0 = in.read_string();
        Account _ob_r = get(_ob_a0);
        out = handler.createReply();
        AccountHelper.write(out, _ob_r);
        return out;
    }

    private org.omg.CORBA.portable.OutputStream
    _OB_op_open(org.omg.CORBA.portable.InputStream in,
                org.omg.CORBA.portable.ResponseHandler handler)
    {
        org.omg.CORBA.portable.OutputStream out = null;
        String _ob_a0 = in.read_string();
        Account _ob_r = open(_ob_a0);
        out = handler.createReply();
        AccountHelper.write(out, _ob_r);
        return out;
    }
}
```

```java
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0
//
public class _AccountManagerStub extends org.omg.CORBA.portable.ObjectImpl
                                implements AccountManager
{
    private static final String[] _ob_ids_ =
    {
        "IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager:1.0",
    };

    public String[]
    _ids()
    {
        return _ob_ids_;
    }

    final public static java.lang.Class _ob_opsClass = AccountManagerOperations.class↙
        ;

    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager/open:1.0
    //
    public Account
    open(String _ob_a0)
    {
        while(true)
        {
            if(!this._is_local())
            {
                org.omg.CORBA.portable.OutputStream out = null;
                org.omg.CORBA.portable.InputStream in = null;
                try
                {
                    out = _request("open", true);
                    out.write_string(_ob_a0);
                    in = _invoke(out);
                    Account _ob_r = AccountHelper.read(in);
                    return _ob_r;
                }
                catch(org.omg.CORBA.portable.RemarshalException _ob_ex)
                {
                    continue;
                }
                catch(org.omg.CORBA.portable.ApplicationException _ob_aex)
                {
                    final String _ob_id = _ob_aex.getId();
                    in = _ob_aex.getInputStream();

                    throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception↙
                        : " + _ob_id);
                }
                finally
                {
```

```
                _releaseReply(in);
            }
        }
        else
        {
            org.omg.CORBA.portable.ServantObject _ob_so = _servant_preinvoke ↙
                ("open", _ob_opsClass);
            if(_ob_so == null)
                continue;
            AccountManagerOperations _ob_self = (AccountManagerOperations)_ob_so.↙
                servant;
            try
            {
                return _ob_self.open(_ob_a0);
            }
            finally
            {
                _servant_postinvoke(_ob_so);
            }
        }
    }
}

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager/close:1.0
//
public void
close(String _ob_a0)
{
    while(true)
    {
        if(!this._is_local())
        {
            org.omg.CORBA.portable.OutputStream out = null;
            org.omg.CORBA.portable.InputStream in = null;
            try
            {
                out = _request("close", true);
                out.write_string(_ob_a0);
                in = _invoke(out);
                return;
            }
            catch(org.omg.CORBA.portable.RemarshalException _ob_ex)
            {
                continue;
            }
            catch(org.omg.CORBA.portable.ApplicationException _ob_aex)
            {
                final String _ob_id = _ob_aex.getId();
                in = _ob_aex.getInputStream();

                throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception ↙
                    : " + _ob_id);
            }
            finally
            {
                _releaseReply(in);
            }
        }
        else
        {
            org.omg.CORBA.portable.ServantObject _ob_so = _servant_preinvoke ↙
                ("close", _ob_opsClass);
            if(_ob_so == null)
                continue;
            AccountManagerOperations _ob_self = (AccountManagerOperations)_ob_so.↙
                servant;
            try
            {
                _ob_self.close(_ob_a0);
```

```
                return;
            }
            finally
            {
                _servant_postinvoke(_ob_so);
            }
        }
    }

    //
    // IDL:hippo.dcs.gla.ac.uk/BankApplication/AccountManager/get:1.0
    //
    public Account
    get(String _ob_a0)
    {
        while(true)
        {
            if(!this._is_local())
            {
                org.omg.CORBA.portable.OutputStream out = null;
                org.omg.CORBA.portable.InputStream in = null;
                try
                {
                    out = _request("get", true);
                    out.write_string(_ob_a0);
                    in = _invoke(out);
                    Account _ob_r = AccountHelper.read(in);
                    return _ob_r;
                }
                catch(org.omg.CORBA.portable.RemarshalException _ob_ex)
                {
                    continue;
                }
                catch(org.omg.CORBA.portable.ApplicationException _ob_aex)
                {
                    final String _ob_id = _ob_aex.getId();
                    in = _ob_aex.getInputStream();

                    throw new org.omg.CORBA.UNKNOWN("Unexpected User Exception
                        : " + _ob_id);
                }
                finally
                {
                    _releaseReply(in);
                }
            }
            else
            {
                org.omg.CORBA.portable.ServantObject _ob_so = _servant_preinvoke
                    ("get", _ob_opsClass);
                if(_ob_so == null)
                    continue;
                AccountManagerOperations _ob_self = (AccountManagerOperations)_ob_so.
                    servant;
                try
                {
                    return _ob_self.get(_ob_a0);
                }
                finally
                {
                    _servant_postinvoke(_ob_so);
                }
            }
        }
    }
}
```

```
// **********************************************************************
//
// Generated by the ORBacus IDL to Java Translator
//
// Copyright (c) 2000
// Object Oriented Concepts, Inc.
// Billerica, MA, USA
//
// All Rights Reserved
//
// **********************************************************************

// Version: 4.0.3

package uk.ac.gla.dcs.hippo.BankApplication;

//
// IDL:hippo.dcs.gla.ac.uk/BankApplication/Currency:1.0
//
final public class CurrencyHelper
{
    public static void
    insert(org.omg.CORBA.Any any, int val)
    {
        org.omg.CORBA.portable.OutputStream out = any.create_output_stream();
        write(out, val);
        any.read_value(out.create_input_stream(), type());
    }

    public static int
    extract(org.omg.CORBA.Any any)
    {
        if(any.type().equivalent(type()))
            return read(any.create_input_stream());
        else
            throw new org.omg.CORBA.BAD_OPERATION();
    }

    private static org.omg.CORBA.TypeCode typeCode_;

    public static org.omg.CORBA.TypeCode
    type()
    {
        if(typeCode_ == null)
        {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init();
            typeCode_ = orb.create_alias_tc(id(), "Currency", orb.get_primitive_tc   ↙
                (org.omg.CORBA.TCKind.tk_ulong));
        }

        return typeCode_;
    }

    public static String
    id()
    {
        return "IDL:hippo.dcs.gla.ac.uk/BankApplication/Currency:1.0";
    }

    public static int
    read(org.omg.CORBA.portable.InputStream in)
    {
        int _ob_v;
        _ob_v = in.read_ulong();
        return _ob_v;
    }

    public static void
    write(org.omg.CORBA.portable.OutputStream out, int val)
    {
```

```
        out.write_ulong(val);
    }
}
```

```
package uk.ac.gla.dcs.hippo;

public class Server
{
    static int run(org.omg.CORBA.ORB orb)
    throws org.omg.CORBA.UserException
    {
        org.omg.PortableServer.POA rootPOA =
        org.omg.PortableServer.POAHelper.narrow(orb.resolve_initial_references
            ("RootPOA"));

        org.omg.PortableServer.POAManager manager = rootPOA.the_POAManager();

        Account_impl accountImpl = new Account_impl();
        Account anAccountObject = accountImpl._this(orb);

        try
        {
            String ref = orb.object_to_string(anAccountObject);
            String refFile = "account.ref";
            java.io.PrintWriter out = new java.io.PrintWriter(new java.io.
                FileOutputStream(refFile));
            out.println(ref);
            out.close();
        }
        catch(java.io.IOException e)
        {
            e.printStackTrace();
            return 1;
        }

        manager.activate();
        orb.run();

        return 0;
    }

    public static void main(String args[])
    {
        java.util.Properties props = System.getProperties();
        props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
        props.put("org.omg.CORBA.ORBSingletonClass",
              "com.ooc.CORBA.ORBSingleton");

        int status = 0;
        org.omg.CORBA.ORB orb = null;

        try
        {
            orb = org.omg.CORBA.ORB.init(args, props);
            status = run(orb);
        }
        catch(Exception e)
        {
            e.printStackTrace();
            status = 1;
        }

        if(orb != null)
        {
            try
            {
                ((com.ooc.CORBA.ORB)orb).destroy();
            }
            catch(Exception ex)
            {
                ex.printStackTrace();
                status = 1;
            }
        }
```

```
        System.exit(status);
    }
}
```