Program Visualisation in an Open Distributed Environment

P. Oldengarm

Informatica

RuG

Afstudeerverslag

Program Visualisation in an Open Distributed Environment

P. Oldengarm

Begeleiders:
Prof.dr.ir. L.J.M. Nieuwenhuis
Dr. J.B.T.M. Roerdink
Rijksuniversiteit Groningen
Informatica
Postbus 800
9700 AV Groningen

maart 1998

Contents

Management Summary

This document reports on a graduate assignment, which was performed for the IT-Lab project (IT-strategies program). The IT-lab provides a highly innovative environment with a state-of-the-art infrastructure for IT services. These services include middleware services. The assignment was supervised by prof dr ir L.J.M. Nieuwenhuis and ir A.T. van Halteren.

Distributed applications, like e.g. middleware applications, are often very complex. It is difficult to explain to other people how these applications work. Program visualisation can help to gain insight in the collaboration structures of these applications. The objective of this graduation project is to answer the question of how to visualise applications in open distributed environments. We want to describe a generic method for visualising distributed applications and apply this method to CORBA applications by means of a visualisation tool. There is a need for such a tool within KPN Research at departments such as CAS, TTS, NSC and SAM.

The document describes a generic method for visualising applications in open distributed environments. The Open Distributed Processing - Reference Model (ODP-RM) has been used for describing the entities that are present in distributed systems. The method describes the visualisation process that has to be performed in four phases: the event collection, event processing, storage and display phases. Furthermore it describes the set of events that can be collected from distributed systems and the set of events that can be visualised.

The generic method is applied in the design and implementation of a visualisation tool for CORBA systems. We present an architecture that handles each of the phases in the visualisation process separately. This architecture has been implemented with Orbix, an implementation of CORBA from Iona Technologies.

This report has relevance to people who wish to demonstrate interactions in distributed applications to other people. If they use Orbix, they can use the tool that we have developed. If they use other environments like e.g. DCOM, they can use our generic method to extend the tool for these environments. This report gives insight in the generic method and the design and implementation issues of the visualisation tool.

1. Introduction

The number of distributed systems is growing. Downsizing mainframe applications to client-server applications has become popular. New technologies for developing distributed applications such as CORBA, Java and ActiveX are in fact turning the Internet into a large distributed object system. These developments provide telecommunication companies with new opportunities, but also with the additional challenges of managing new paradigms for developing and deploying services.

Developing distributed systems is a complex task. Tools and modelling languages such as the Unified Modelling Language (UML) are helpful, but lack ways for expressing distribution of objects over the network. Therefore, a clear understanding of the collaboration and distribution of objects has become a difficult task. Program visualisation can help gaining insight in how a system works and could be used to discover or prevent potential bottlenecks. In addition, program visualisation can expose design errors in the distribution strategy of an object system.

1.1 Objective

In this thesis we want to answer the question of *how to visualise applications in open distributed environments.*

First we want to describe a generic method for visualising applications in open distributed environments. The ISO/ITU-T International Standard for the Reference Model for Open Distributed Processing (ODP-RM) is used for defining what has to be visualised, but also for describing the information that can be extracted from a distributed system. This contributes to a clearer understanding of the relation between CORBA and ODP concepts.

Secondly we want to apply this method in the design and implementation of a tool for the visualisation of CORBA systems. Before designing such a system, the appropriate entities and levels of abstraction must be identified. In addition to this, a mechanism must be designed and implemented to catch events from a CORBA system and to translate them into events that have to be visualised. In this mechanism a correlation has to be made with the design issues of the CORBA system that is visualised.

1.2 Thesis Structure

Because ODP-RM is used for describing the distributed systems we want to visualise, we first study the principles of this reference model. The systems on which we want to apply our generic method are CORBA systems. Therefore we also study the basics of CORBA and take a look at the relationship between ODP-RM and CORBA. This is described in Chapter 2.

Next we need to study the principles of program visualisation. Chapter 3 gives a classification of visualisation and describes the aspects that have to be considered when designing user interfaces.

The third step is to take a close look at articles that have been published about program visualisation of open distributed systems. We make a comparison with program visualisation of parallel software systems. This is described in Chapter 4.

Based on the principles of Chapter 2, 3 and 4, we present a generic method for visualising applications in open distributed environments. We have applied this method to CORBA systems in the design and implementation of a visualisation tool. This tool has been evaluated by future users of the tool. These subjects are described in Chapter 5.

The last part of this thesis, Chapter 6, presents the conclusions. We give recommendations for extending the tool. Furthermore we give recommendations towards the Object Management Group (OMG) for the standardisation of visualisation events.

2. Open Distributed Systems

This chapter gives an overview of the Reference Model of Open Distributed Processing (ODP-RM). It describes the motivation for a standard in open distributed processing and the concepts of the model. This chapter also gives an introduction to the basic concepts of CORBA. In the last section of this chapter the relationship between CORBA and ODP concepts is discussed. The objective of this chapter is to provide the necessary information on ODP-RM and CORBA that is used as a basis for the remaining part of this thesis.

2.1 The Open Distributed Processing Reference Model (ODP-RM)

The Open Systems Interconnection Reference Model (OSI-RM) gives a standard for protocols, needed for interconnecting systems. This reference model is restricted to communication standards and does not go into the matter of distributed applications. That is why in 1987 a new standardisation activity was initiated, called Open Distributed Processing. The ODP Reference Model provides a standardisation framework for distributed systems.

The ODP-RM provides concepts, rules and models for building distributed systems. The main objective of ODP is to enable the interworking between heterogeneous distributed systems and applications. The ODP-RM defines several important foundations that are used throughout the standard. ODP-RM uses two abstraction mechanisms to deal with the complexity of distributed systems: *distribution transparencies* and *ODP viewpoints* [Leyd 1997], [Nank 1996], [ODP-1 1995], [ODP-2 1995], [ODP-3 1995], [ODP-4 1995].

2.1.1 Distribution transparencies

A distribution transparency is the property of hiding the behaviour of some parts of a distributed system to a particular user. Table 2-1shows the distribution transparencies as defined in ODP-RM [ODP-3 1995].

Transparency	Masks
Access	Masks the difference in data representation and invocation mechanisms (i.e. protocols) to enable interworking between objects.
Failure	Masks the failure and possible recovery of other objects (or the object itself) to enable fault tolerance.
Location	Masks the use of location information from actions that make a relation between the interfaces of two objects, using a communication path (binding actions).
Migration	Masks the ability of a system to change the location of an object.
Relocation	Masks the relocation of an interface from other interfaces bound to that interface.
Replication	Masks the use of replicated objects in support of an interface.
Persistence	Masks from an object the activation and deactivation of other objects (or the object itself).
Transaction	Masks the co-ordination of activities amongst a configuration of objects to achieve consistency.

Table 2-1: Distribution Transparencies

2.1.2 ODP Viewpoints

A distributed system can be quite complex. To deal with this complexity, ODP uses a framework in which the system is viewed from five different viewpoints. Each viewpoint represents a different view on the system with emphasis on a specific aspect. It abstracts from the aspects that are irrelevant for that viewpoint.

ODP distinguishes the following viewpoints: Enterprise Viewpoint, Information Viewpoint, Computational Viewpoint, Engineering Viewpoint and Technology Viewpoint (see Figure 2-1).

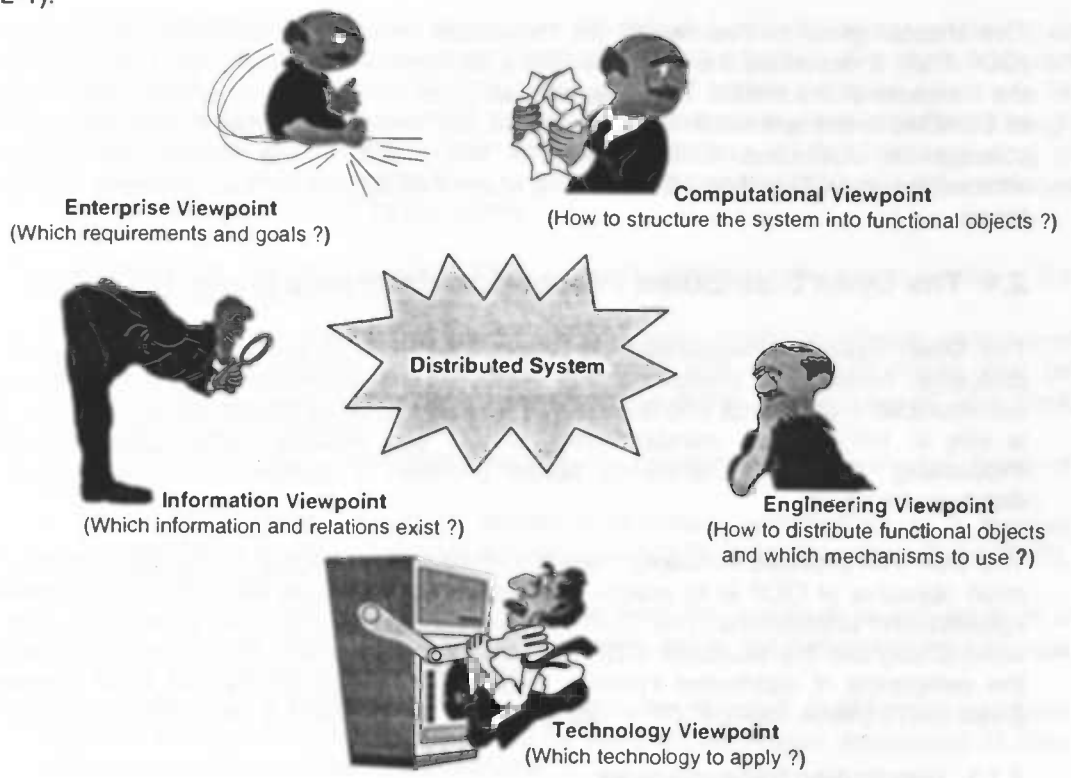


Figure 2-1: ODP-RM Viewpoints

These five viewpoints can be characterised as follows:

- The *Enterprise Viewpoint* focuses on the requirements, purpose and policies of the ODP system. The enterprise specification describes the overall objectives of the system in terms of roles, actors, goals and policies. Its purpose is to explain and justify the objectives of the system used by one or more organisations.
- The *Information Viewpoint* identifies and locates the information within the system under concern. It describes the flows of information in the system. The main concern is the syntax and semantics of the information and information flows within the system.
- The *Computational Viewpoint* provides a functional decomposition of the system. It describes how the objects in the system interact in a distribution transparent way. ODP defines eight distribution transparencies (see Table 2-1). Distribution transparency means that the structuring of the applications does not depend on the computers and networks on which they run.
- The *Engineering Viewpoint* focuses on the infrastructure required to support distribution of the system. It is concerned with the distribution of objects and with the provision of various mechanisms to support distribution.
- The *Technology Viewpoint* identifies the physical components, both hard- and software, required for realising an ODP system. It is concerned with the implementation details of the components in the distributed system.

2.1.3 Viewpoint Languages

In order to represent an ODP system from a particular viewpoint it is necessary to define a structured set of concepts to make a specification in that viewpoint. This set of concepts provides a language for writing specifications of the system from that viewpoint. The language is the means to create a model of the problem domain from the concerned viewpoint.

For each viewpoint a language is defined for writing specifications of ODP systems. In the following sections we summarise the principles of each language. For a more detailed description we refer to Leydekkers and the description of the standard [Leyd 1997], [ODP-1 1995], [ODP-2 1995], [ODP-3 1995], [ODP-4 1995].

Enterprise Language

The enterprise language contains concepts to represent an ODP system in terms of interacting agents, working with a set of resources to achieve business objectives subject to the policies of controlling objects.

An enterprise model is defined in terms of:

- *Roles* that are played by the enterprise objects involved for the purpose of achieving an objective.
- The *community* concept is used for grouping enterprise objects to meet a common objective
- The *policies* set down rules for the permission of actions by certain objects.
- A *contract* links enterprise objects that perform a certain role and expresses their mutual obligations.
- A *federation* is a special kind of community. It is defined as the property of combining systems from different administrative or technical domains to achieve a single objective. The creation of federations and the expression of the rules to control federations forms an important part of the system specification in the enterprise viewpoint.

Information Language

The information language contains concepts to enable the specification of the meaning of information manipulated by and stored within ODP systems. The information held by the system is described in terms of information objects, and their relationship and behaviour.

An information specification consists of a set of related schemata. ODP-RM defines three types of schemata:

- A *static schema* describes the state and structure of an information object at some particular interesting situation. For instance, it might be used to specify the initial state of an object.
- An *invariant schema* expresses relationships between objects which must always be true, for all valid behaviour of the system.
- A *dynamic schema* describes how the information can evolve as the system operates. It specifies the permitted changes in state and structure of an object subject to the constraints in the invariant schema. In addition to describing state changes, dynamic schemata can also create and delete information objects.

An *information template* in ODP consists of a combination of static, invariant and dynamic schemata.

Computational Language

In the computational viewpoint, a distributed application consists of a collection of computational objects. A computational object provides a set of services that can be used by other objects. An object offers a computational interface to enable other objects to access its service.

The computational language is used to describe the structure of a distributed application and specifies the interaction between computational objects. It describes the service and behaviour of the distributed application.

The computational viewpoint assumes distribution transparencies (see Table 2-1). Therefore, the computational language hides the degree of distribution of an application from the specifier. This implies that applications make no assumptions about the location of their components.

The objective of the computational language is to make a functional decomposition of an ODP system into objects that interact with each other at an *interface*. The objects interact according to the client/server model. The server provides certain services, which can be requested by the clients. The services provided by the servers are accessible through interfaces. A server object can support multiple interfaces, which allows grouping of related services. If the client wants to use a service of a server, it requires the interfaces at which this service is offered by the server.

The computational language provides three types of interfaces: the signal interface, the operational interface and the stream interface.

- *Signal interfaces* are elementary interfaces. The signal interface initiates signals and might receive responding signals. Both operation and stream interfaces can be decomposed into signal interfaces. An example of a signal interface would be a clock interface, emitting a pulse once every second.
- All interactions at the *operational interface* are operations. There are two types of operations:
  1. At an *interrogation* a server returns a response (termination) to a request (invocation) of a client. This style is very similar to Remote Procedure Calls (RPC), or (remote) method calls of objects.
  2. In an *announcement* there is no response from the server at a request from a client. This can be compared with a procedure call in the programming language Pascal (without any output parameters).
- All interactions at a *stream interface* are continuous flows of data from a producer object to a consumer object. Flows may be used for continuous sequences of data transmissions between clients and servers.

Engineering Language

The engineering language describes the way that object interaction is achieved and the resources needed to do so. In the computational viewpoint the main concern is *when* and *why* objects interact, whereas the engineering viewpoint focuses on *how* object interaction is achieved.

In the engineering language the computational objects are visible as *Basic Engineering Objects* (BEOs) and the bindings are visible as *channels* or local bindings. The engineering language has three basic concepts: the node components, the interactions of nodes and the functions provided at engineering level. We describe each of these concepts in the following paragraphs.

Figure 2-2 shows how engineering objects are structured. A *node* is an engineering abstraction of a (physical) computing system. Nodes can be regarded as autonomously managed computing systems. Every node is under control of a *nucleus*. A nucleus is the engineering abstraction of an operating system. It is an object that co-ordinates processing, storage and communication functions used by other engineering objects within the same node.

A *capsule* is an isolated subset of the resources of a node. It owns storage and a share of the node's processing resources. It can be compared to a UNIX process with its own address space. The *capsule manager* is an object associated with each capsule. A capsule is controlled by interactions with this manager.

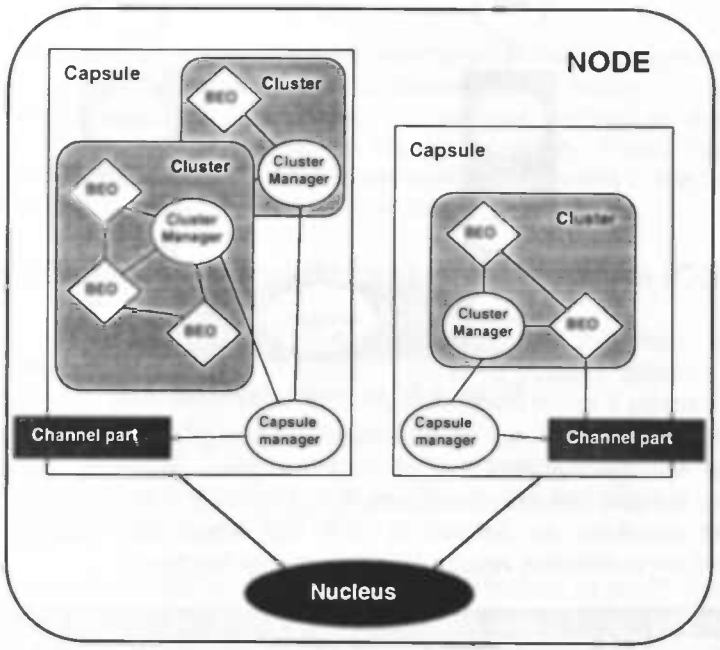


Figure 2-2 : Engineering concepts

A capsule can contain many objects. Grouping of objects in a capsule reduces the cost of object interaction. The smallest grouping of objects is a *cluster*. Objects in a cluster are grouped together to reduce the overhead to manage them (e.g. instantiate, delete the objects, etc.). Clusters are controlled and actions on them initiated by interaction with an associated *cluster manager* object.

There are two types of engineering binding between engineering objects. The first is the binding between objects that are grouped in one cluster. The binding between those objects is called local binding and is resolved by system specific mechanisms. The second is the binding between objects that are not grouped in one cluster. To bind these objects a *channel* is needed. A channel is a configuration of *stub*, *binder*, *protocol* and *interceptor* objects that interconnect a set of Basic Engineering Objects (see Figure 2-3).

A stub is an object that provides conversion functions for data, exchanged between two or more BEOs. A stub provides wrapping and coding functions for the parameters of an operation. This also referred to as *marshalling* [ODP-3 1995].

A binder object maintains a binding among interacting engineering objects. It manages the end-to-end integrity of the channel and deals with object failures. The binder object is involved in many of the distribution transparencies as described in Table 2-1.

A protocol object makes it possible that BEOs can interact (remotely) with each other. The protocol objects are needed if the computational objects that have to be bound are located in different nodes.

An interceptor is an object at a boundary between domains. Interceptors play a role if interacting protocol objects are in different domains. It can be used to enforce security policies.

The stub, binder or protocol object itself may need to communicate with other parts of the system, in order to obtain the information it needs to perform its task, or to supply management information to other objects.

Any of these objects can support control interfaces, via which they can be managed.

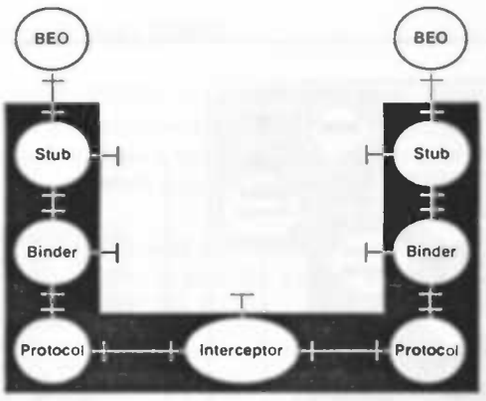


Figure 2-3 : An engineering channel

ODP-RM defines a number of common functions at engineering level which are fundamental for the construction of ODP conform systems. Several functions are introduced to support various transparencies as defined in Table 2-1. Table 2-2 shows the collection of functions as defined in ODP-RM which are classified in four groups: *management*, *co-ordination*, *repository* and *security* functions.

Function category	Function name	Role
Management	node manager	controls processing storage and communication functions within a node
	capsule manager	instantiates, checkpoints and deletes all the clusters in a capsule and deletes capsules
	cluster manager	checkpoints, recovers, migrates, deactivates or deletes clusters
	object manager	checkpoints and deletes objects
	migration	co-ordinates the migration of a cluster from one capsule to another
	checkpoint recovery	co-ordinates the checkpointing and recovery of failed clusters
	deactivation/ reactivation	co-ordinates the deactivation and reactivation of clusters
	event notification	records and makes available event histories
Co-ordination	transaction	co-ordinates and controls a set of transactions to achieve a specified level of visibility, recoverability and permanence
	ACID transaction	special case of transaction
	replication	ensure a group appears to other objects as it were a single object.
	Group	co-ordinates the interactions of objects in a multi party binding
	engineering interface reference tracking	monitors the transfer of engineering interface references between engineering objects in different clusters
	relocator	manages a repository of locations for interfaces
	storage	stores data
Repository	type repository	manages a repository of type specifications and type relationships
	trading	mediates advertisement and discovery of interfaces
	information organisation function	manages a repository of information
	access control	prevents unauthorised interaction with an object
	security audit	provides monitoring and collection of information about security related actions
	authentication	provides assurance of the claimed identity of an object
Security	integrity	detects and/or prevents the unauthorised creation, alteration or deletion of data
	non repudiation	prevents the denial by one object of having participated in all or part of the interaction
	key manager	provides facilities for the management of cryptographic keys
	confidentially	prevents the unauthorised disclosure of information

Table 2-2 : Engineering functions

Technology Language

The technology language describes the implementation of the ODP system in terms of a configuration of objects representing the hardware and software components of the implementation. It is constrained by cost and availability of technology objects (hardware and software products) that would satisfy the specification. These may conform to implementable standards. Thus, the technology viewpoint provides a link between the set of viewpoint specifications and the real implementation.

2.2 The Common Object Request Broker Architecture (CORBA)

At the beginning of the nineties a few IT companies realised that there would never be consensus on which programming language, operating system, processor technology, etc. they would all use. They needed something that would make it possible that software in heterogeneous environments could co-operate. That is why the Object Management Group (OMG) was founded. Their goal was to standardise the *interoperability* in heterogeneous software systems. This has leaded to the specification of a Common Object Request Broker Architecture: CORBA.

The OMG has since its foundation grown into an organisation of world concern and over 800 IT companies, representing the entire spectrum of computer industry, support OMG. A notable exception is Microsoft, which develops its own object broker called the Distributed Component Object Model: DCOM [OMG-1 1997], [Orf-1 1997].

In the following sections we discuss the main principles of CORBA: the *Interface Definition Language*, the *Object Request Broker* and the *CORBA Services* and *Facilities*. For a more extensive description of the CORBA standard we refer to [Orf-1 1997], [Orf-2 1997] and [OMG-2 1997].

2.2.1 The Interface Definition Language (IDL)

OMG uses the Interface Definition Language (IDL) to describe objects in the system. The special thing about IDL is, that it only describes the "outside" of the object and that it doesn't say anything about the implementation of the object. That leaves the choice of programming language and operating system to the programmer. IDL only defines the *interface* to the object. An object can only be accessed through methods that have been defined in the object's interface.

IDL provides operating system and programming language independent interfaces to all the services and components that reside on a CORBA bus. It allows client and server objects written in different languages to interoperate (see Figure 2-4).

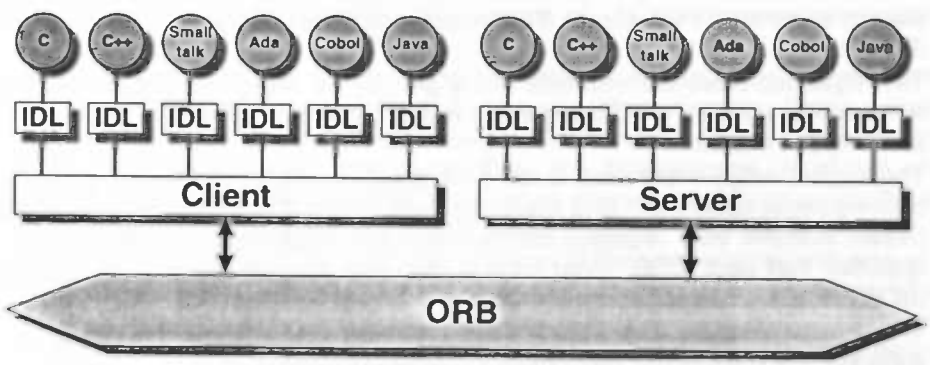


Figure 2-4 : CORBA IDL Language bindings provide client/server interoperability

2.2.2 The Object Request Broker (ORB)

The Object Request Broker (ORB) is a software bus interconnecting IDL specified objects. The ORB handles all data transfers between objects. The objects can either be clients or servers. The client makes a request on a server and the server gives a reply to the client. The server can also act as a client and request a service from another server.

The ORB consists of several (software) components shown in Figure 2-5. The ORB can be divided in two parts: the client side and the server side. In the following paragraph we describe the components of the CORBA ORB.

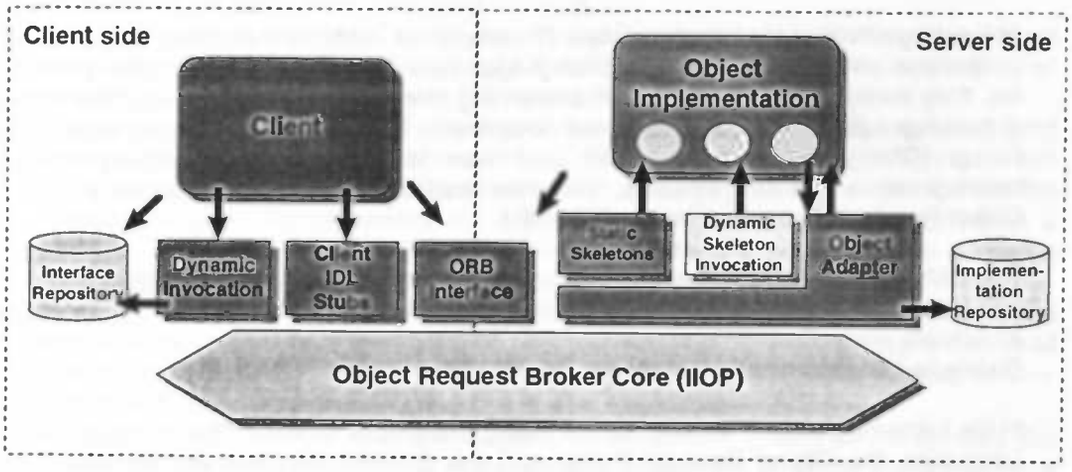


Figure 2-5 : The structure of a CORBA 2.0 ORB

On the client side the following components reside:

- The *Client IDL Stubs* provide the static interfaces to object services. These precompiled stubs define how clients invoke corresponding services on the servers. A client must have an IDL stub for each interface it uses on the server.
- The *Dynamic Invocation Interface* (DII) lets you discover methods to be invoked at run time.
- The *Interface Repository* contains the descriptions of all the registered component interfaces, the methods they support and the parameters they require.
- The *ORB Interface* consists of a few Application Programming Interfaces (APIs) to local services that may be of interest to an application. E.g. methods are provided to transform a object reference into a string or to retrieve the interface of an object (Naming Service).

The server side contains the following elements:

- The *Server IDL Stubs* (OMG calls them skeletons) provide static interfaces to each service exported by the server. These stubs, like the client stubs, are created using an IDL compiler.
- The *Dynamic Skeleton Interface* (DSI) provides a run time binding mechanism for servers that need to handle incoming method calls for components that do not have precompiled skeletons.
- The *Object Adapter* provides a run time environment for instantiating server objects, passing requests to them and assigning them object IDs, called *object references*. The Object Adapter also registers classes with the Implementation Repository. CORBA specifies that each ORB must have a standard adapter called *Basic Object Adapter* (BOA). Servers can support more than one Object Adapter.
- The *Implementation Repository* contains information about the classes a server supports, the objects that have been instantiated and their IDs.
- The *ORB Interface* is identical to the ORB Interface on the client side of the ORB.

2.2.3 The CORBA Services and Facilities

The work of OMG isn't restricted to the software bus. OMG has also developed object services and facilities.

The Object Services are standard components that are specified in IDL. They can be used in all kinds of applications. OMG has published standards for fifteen object services. A few of these services are listed below.

- The *Life Cycle Service* defines operations for creating, copying, moving and deleting components on the bus.
- The *Persistence Service* provides a single interface for storing components persistently on a variety of storage servers.
- The *Naming Service* offers location transparency. This implies that components in the system don't have to know the location of other components. The components are located by name. These logical names are mapped to Interoperable Object References (IOR's).
- The *Event Service* defines an *event channel* that collects and distributes events among interested objects. An object can (un)register its interest in specific events.
- The *Security Service* allows identification and authentication of users of objects, authorisation and access control of objects, administration of security information, etc.
- The *Transaction Service* provides two phase commit co-ordination among recoverable components using either flat or nested transactions.

The Transaction service and the Persistence service implement two distribution transparencies as they are described in ODP-RM (see Table 2-1).

The CORBA Facilities are more advanced application components that are aimed towards a certain branch of industry. Contrary to the services, the facilities are only applicable in a restricted set of applications. The Common Facilities that are currently under construction include mobile agents, data interchange, business object frameworks and internationalisation.

2.3 The relationship between CORBA and ODP concepts

It is convenient to use ODP-RM for describing a distributed system that has to be visualised for different target groups of users. The reason is that ODP gives a generic framework for Open Distributed Systems. Furthermore ODP-RM provides a way to abstract from details that are not relevant in a certain view on the system.

Our visualisation tool has to visualise CORBA systems. Therefore we apply the ODP concepts to CORBA systems. CORBA can be interpreted as an *instance* of the *class of ODP systems*. Since we use ODP-RM to describe how a CORBA system is visualised, we have to relate ODP-RM concepts to aspects in CORBA. In principal all ODP viewpoint languages are candidates for describing the aspects of a CORBA system to visualise.

CORBA implementations have the following characteristics. Clients and servers are (part of) processes that run on machines with certain operating systems. Those clients and servers communicate with each other through TCP/IP connections (when the Internet Inter ORB Protocol (IIOP) is used) and they are programmed in object oriented programming languages, like C++ and Java, or in conventional languages like C and COBOL. They consist of a group of objects that are written in those languages. These characteristics find a detailed match to the concepts of the engineering language of ODP-RM. The engineering language is the most elaborate viewpoint language, with which we can describe a large set of concepts without losing genericity. This is in contrast with the other viewpoint languages. With the enterprise language e.g. we can only describe a small part of the concepts that are present, because the enterprise viewpoint abstracts from too many issues. The technology language e.g. is limited in size and not very generic, so it does not provide a good base for describing what happens in a CORBA system. The computational and information language do not explicitly deal with the physical distribution of objects, so they are not suitable for describing aspects of

visualisation of CORBA systems. Therefore we have chosen the engineering language for describing what happens in a CORBA system.

For the concepts defined in the engineering viewpoint language we give a mapping to aspects that can be identified in the CORBA system.

- A *node* in ODP-RM is a physical abstraction of a (physical) computing system. So the node is the *machine* on which objects in a CORBA system runs. A machine is also called a *host*.
- Every node in an ODP system is under control of a *nucleus*. A nucleus is an abstraction of the *operating system*. A CORBA system can run under UNIX or Windows NT, or many other operating systems.
- A *capsule* can be compared to an operating system process with its own address space. An important feature of a capsule is that when it fails, it should not affect other capsules. On every node a capsule has a unique process id.
- A *capsule manager* controls the interactions in a capsule. There is no generic mapping from a capsule manager to an aspect in a CORBA system. It is possible to implement a capsule manager in a CORBA system with a *factory object*, or an API to the operating system that organises the instantiation and deletion of objects within a capsule. In this way it controls the interactions within a capsule. This is a design specific issue.
- A capsule can be decomposed into parts that perform a certain function. These parts are called *clusters* in ODP-RM. Objects are grouped into clusters to reduce overhead to manage them individually. A designer can e.g. define a cluster as a group of C++ objects and object implementations. This is also a design specific issue.
- Clusters are controlled by *cluster managers*, which perform actions on the clusters like instantiation and deletion. There is no generic equivalent of these managers in a CORBA system. They can be implemented by *administration objects* that keep track of the objects in a cluster. Therefore, the cluster managers are design specific.
- The objects in a cluster are the basic elements of the system and they are called *basic engineering objects* (BEOs). In CORBA these objects are instances of single classes (e.g. C++, Java, Smalltalk classes, or object implementations).
- Objects in different capsules communicate through *channels*. In a CORBA system using IIOP a channel is a *TCP/IP connection* between two processes.

Table 2-3 summarises the relationships that we described in the previous paragraph.

ODP	CORBA
Node	Machine
Nucleus	Operating System
Capsule	Operating System Process
Capsule Manager	*design specific*
Cluster	*design specific*
Cluster Manager	*design specific*
Basic Engineering Object	C++ Object
Channel	TCP/IP Connection

Table 2-3: Relationship between CORBA and ODP

3. Principles of Program Visualisation

This chapter gives an overview of the principles of program visualisation. We start with a classification of visualisation into various subfields (Section 3.1). In Section 3.2 we give some general thoughts collected from the literature about how to make a good visual representation.

3.1 Classification

Increasingly, graphical visualisations are becoming useful and powerful tools for understanding complex tasks. These visualisations can be divided in three area's [Tomas 1994]:

- Scientific Visualisation: using graphical representations of data to gain insight in the structure of the data.
- Program Visualisation: using visualisation to gain insight in the behaviour of a program.
- Visual Programming: specifying a program in a two dimensional graphical form.

We are interested in the second item: Program Visualisation. Program Visualisation can be used for many purposes. In addition to gaining insight in the operation of a program, it can be used for debugging purposes or performance evaluation.

3.2 Good visual representations

In general, there is no fixed set of rules to follow when creating a visualisation. However, many articles and books have been written about improving visual representations. In the following paragraphs we discuss a selection of the ideas that are presented in these articles and books.

3.2.1 Drawing of graphs

In this thesis, a tool for visualisation of a distributed system is made. This system is represented by a graph. Thus the first visualisation aspect of interest is how to make nice drawings of graphs. This subject is discussed by Brandenburg [Bran 1988].

Brandenburg claims that the main quality desired for diagrammatic representations is readability. A diagram is readable in his opinion if its meaning is easily captured by the way it is drawn. The pictorial representation should focus our view to the more important parts of the drawn object and should illustrate its global structure. This, however, is imprecise and depends on various features, including the intended meaning of the diagram. The problem is approached by using *graph embeddings*, a mathematical model to compare features of graphs. These features include planarity of the graph, hierarchy, cycles, etc. The parameters in the model depend on the features in the specific visualisation and have to be added by the individual user, according to his/her personal needs. The goal is choosing an embedding that optimises the parameters under consideration. The parameters are e.g. expansion (size of the embedded graph), area (size of the smallest enclosing rectangle on the grid), or the crossing number for non-planarity.

In our case, the problem is that vertices and edges in the graph appear and disappear dynamically. This means that a *nice* drawing at a certain location in time can change into a *bad* drawing at a later time. In my opinion it would be best to create a good drawing at

the start of the visualisation and give the user of our tool the possibility to change the location of the vertices during operation.

3.2.2 Colour

Colour can contribute a lot to the understanding and view of a visualisation. It can also damage it by diverting one's intention from the essential parts of the visualisation. Therefore it's important to give a moments of thought as on how colour should be used in the visualisation. We present some major thoughts here as they were described in [Tufte 1990], [Brown 1991] and [Fowler 1995].

Tufte presents four rules for avoiding *colour damage* in a visual representation:

Rule 1 Pure, bright or very strong colours have loud, unbearable effects when they stand unrelieved over large areas adjacent to each other, but extraordinary effects can be achieved when they are used sparingly on or between dull background tones.

Rule 2 The placing of light, bright colours mixed with white next to each other usually produces unpleasant results, especially if the colours are used for large areas.

Rule 3 Large area background or base colours should do their work most quietly, allowing the smaller, brighter areas to stand out most vividly, if the former are muted, greyish or neutral.

Rule 4 If a picture is composed of two or more large, enclosed areas in different colours, the picture falls apart. Unity will be maintained, however, if the colours of one area are repeatedly intermingled in the other and/or if the colours are interwoven carpet fashion throughout the other.

A better visualisation is created if these rules are applied. Additionally Tufte gives the advise to use natural colours because they are familiar and coherent and they have a widely accepted harmony to the human eye.

While Tufte's rules apply to all kinds of visualisations (e.g. drawing of maps), Brown gives some general ideas about the use of colour in an animation. Brown states that colour has the potential to communicate lots of information efficiently; however, it is not easy to achieve this goal. He tries to apply the principles of Tufte in animation systems and uses colour for five purposes:

1. to encode the (state of) data structures
2. to tie different views on the animation together
3. to highlight activity
4. to emphasise patterns
5. to make history visible

All of these principles apply to the visualisation system we want to build. We discuss each of them in the following paragraphs.

When the (state of) data structures is encoded with a certain colour, you recognise immediately groups of structures that have the same properties. It gives a better understanding of which parts in the system have the same functionality.

In different views on the system, the same colour for the same elements should be used. This creates a feeling to the user that he/she has already seen these items before. It therefore gives a sense of consistency and a better insight in the structure of the program that is visualised.

When activity in the system is highlighted by a colour, the user is immediately aware of the interesting parts in the visualisation. He/she doesn't miss the crucial parts of the visualisation if this technique is used.

If a certain object in the visualisation is given a colour that fades away in a certain period of time, a sense of history is created. In this way, a user can see what the ordering of the actions in the visualisation is.

Fowler gives four reasons to use colour in a graphical user interface:
1. Colour is more interesting than black and white.
2. Colour can alert users to problems or to changes in system states quickly.
3. Colour coding shows relationships quickly.
4. Colour coding shows differences quickly.

Fowler states that before you use colour to code your visualisation, you should take a look at the following colour coding rules:
1. Colour coding is useful only if the user knows the colour code.
2. The advantage of colour increases as clutter in a display increases. I.e. when there are many objects in a display, colour can be used to distinguish them. In a complicated, high density display (60 items), colour can reduce search time by 90 percent. However, when there is too much clutter in a display, colour adds nothing to performance.
3. Average search time increases linearly as the numbers of items using the same colour increases. In other words, you lose some of your colour advantage if too many items have the same colour.
4. Items that don't use the target colour have no effect on search time if their colour is sufficiently dissimilar from the target colour. E.g. since red is very different from yellow, no user will pick a yellow triangle while looking for a red triangle.

Fowler gives in her book some tips on how to use colour in a graphical user interface. These tips are similar to the tips given by Tufte, which we described at the beginning of this paragraph.

3.2.3 Animation

Little attention in the literature is devoted to the techniques that a visualisation must use to design dynamic graphics [Brown 1991]. In his article Brown reviews the techniques he and Sedgewick developed in 1984 [Brown 1984]. We summarise the techniques that are important for our visualisation.

State cues — Changes in the state of data structures should be reflected on the screen by changes in their visual representations. For example, a round object can change into a square when its state changes. State cues link different views of the system together and they reflect the dynamic behaviour in a visualisation.

Amount of input data — If the amount of input data is low, the visual representation gives a clearer view on the system. Not too many objects should be viewed at the same time.

Continuous versus discrete transitions — When a change to a data structure is represented graphically, the change may either be continuous or discrete. Continuous change is most helpful for small data sets; for large enough amounts of data, small discrete changes look smooth, and any smoother motion would not be noticeable.

3.2.4 Graphical User Interface Components

The design of good graphical user interface (GUI) components is described in [Fowler 1995]. In this paragraph we summarise the ideas given in this book with respect to windows, buttons and pointers: the components we use in our tool.

Windows
Application windows can be divided into three functional types:
- *Form based data entry windows*, in which the user types data from a paper form into the computer.

- *Conversational windows*, in which the user interacts with the computer.
- *Inquiry windows*, in which the user searches and retrieves specific information.

In our tool we use a conversational window. The users look at and interact with the window itself. Fowler states that since the user's attention is on the window, more information is better than less. Conversational windows can become crowded, however, and highly complex. Without careful design, they can become confusing. Fowler gives a simple test for whether a conversational window has been designed correctly:

> *"Can all screen elements (field labels, data, title, headings, etc.) be identified without reading the words that make them up?"*

Designing windows that pass this test takes work. According to Fowler it is not hard if you:

- Lower the density of elements in the window.
- Align fields and labels well
- Write labels correctly.

The overall density of a conversational window should be between 25 and 30 percent. In other words, 70 to 75 percent should be empty space. This is not wasted space, however, the blank areas draw the user's eye to what you want him or her to notice. Fowler gives the following recommendations to simplify a complicated window:

- *Organise the information:* Put the most often entered or referenced information at the top and the least often used information at the bottom or in dialogue boxes.
- *Position buttons correctly:* Put buttons related to the entire window at the bottom of the window. Put buttons related to sections of the windows inside those sections.
- *Create functional groups of information:* Put parts that belong together in groups. Break up the groups by putting them in boxes or by separating them with blank lines or rules. If there are no functional breaks, then break the screen every five to seven rows.
- *Provide only need-to-know information:* Put the most important information at the top of the window. Less important information can go at the bottom of the window.
- *Put nice-to-know information in dialogue boxes:* Al the important information should be in the main window. Information that is nice-to-know can appear in dialogue boxes.

Aligning the fields in a window can reduce the complexity. To reduce the complexity of a window we can minimise the number of rows and columns in a window. Fowler gives a formula to test the complexity of a window:

Complexity = # (fields, labels, titles, buttons, etc.) + # rows + # columns

The best solution is the window with the lowest complexity.

The last subject we discuss in this paragraph is how to write labels. Labels do not merely have to look good, they have to read well, too. Recommendations for writing labels are:

- Use symbols like $, #, % only if all users will understand them.
- Try to use short, familiar words. As well as being more readily understood, short words tend to be more authoritative. However, a long, familiar word is better than a short, unfamiliar one.
- Try to use positive terms, which are generally better understood than negative terms.
- Don't stack letters to label a column or a table:

  C
  o
  l
  u
  m
  n

  Instead, put the label above the column or table or turn the entire word sideways.
- For better readability, don't break words between lines.

There are also some rules where to put labels on a screen and what they should look like. Field labels should be a word or phrase followed by a colon and a space:

Label:_Data

Put labels for columns above the columns and labels for the individual fields in front of the fields. Don't put labels *above* individual fields. As well as using two lines per field, instead of one, the labels tend to become visually detached from their fields. Furthermore labels should be as close as possible to their fields and line up vertically in organised columns.

Buttons

GUI development kits include the following types of buttons:

- Push buttons, which let users take actions.
- Radio buttons, used for mutually exclusive choices.
- Check buttons, usually used for settings.
- Sliders and spin buttons, which let users select points on ranges or select from lists.

We discuss the design principles of these kinds of buttons in the following paragraphs.

Push buttons are used for designating, confirming, or cancelling an action and are therefore also known as *action* and *command* buttons. The labels on push buttons must unambiguously identify its use. Since push buttons lead to actions, their labels are usually verbs, either text, or symbolic.

Radio buttons (also called *option buttons*) are used for mutually exclusive choices. In any set of radio buttons, only one can be pushed in. Radio buttons have two labels. One for the overall set of buttons and labels for the individual buttons.

Check buttons are used for multiple, not mutually exclusive choices. Check buttons, like radio buttons, have two labels: a label for the overall set of choices and labels for the individual buttons.

Sliders and spin buttons are used to represent a value on a scale. The scale can be represented with a set of markers, a percentage or a numerical value. A slider arm can be replaced with spin buttons, which are small squares with up and down or left and right arrows. Spin buttons are used to display long lists of choices that increase or decrease in constant units.

Pointers

Pointers have two components: a visual representation (arrow, question mark, etc.) and a hotspot. The hotspot is an area inside the pointer that marks the exact location on the desktop that will be affected by the user's next mouse action.

Fowler gives a few design recommendations for the design of pointers:

- The shape of the pointer should give some hint as to its purpose.
- The shape should be easy to see and recognise. Users may be unable to understand the pointer if the image is too small and the details are too fine or if the image is not familiar enough.
- Avoid visual clutter. A pointer is a small element on a large screen. If you cram too much detail into a tiny space, the users won't be able to figure out what they are looking at (see Figure 3-1).
- The hotspot should "feel" obvious. Put the hotspot at the tip of an arrow, not at the end.

Good Poor

Figure 3-1: Clutter in a pointer

3.3 Conclusion

Program Visualisation is a powerful tool to give other people a better understanding of the operation of a program. We use program visualisation to explain the operation of distributed systems for different target groups.

The principles of this section are applied to the graphical user interface of our visualisation tool, which is described in Chapter 5.

4. Visualisation state-of-the-art

This chapter gives an overview of the research that has been done on the visualisation of Open Distributed Software Systems. We make a comparison with the visualisation of Parallel Software Systems and describe the advantages and disadvantages of the various tools. The objective of this chapter is to give an overview of previous research in this area.

4.1 Parallel Software Systems

For many years, research has been done on parallel systems. This includes research on visualisation of parallel systems. A parallel system is a system in which the processes that run on different nodes, work together to accomplish a common goal. To achieve this goal, they communicate with each other to exchange information. This is similar to distributed systems in which processes that can run on different machines, communicate with each other. The difference between a parallel system and a distributed system can be described as follows. In a parallel system the same program runs on every node. Every node performs the same actions on a little part of a large data set. In a distributed system on every node a different program runs that performs a task that the node is good at. So to display images, e.g. a Silicon Graphics machine could be used and to process database information, a Database server would be useful.

Because the research field of parallel systems is much older than the field of open distributed systems, we first study the principles of visualisation of parallel programs.

In [Krae 1993] Kraemer gives an overview of visualisation tools for parallel systems. The article serves as a bibliographic summary of existing research on the visualisation of parallel systems. The focus in the article is on parallel debuggers, performance evaluation and program visualisation systems. Kraemer characterises research and systems on two levels: the visualisation task being performed and the purpose of visualisation. The tasks include:

1. data collection
2. data analysis
3. storage of data
4. display

The purpose of a visualisation may be to debug a program or system, to evaluate and optimise performance or program visualisation. The latter is the most interesting for our research. We describe for each task the principles, that are of importance for program visualisation.

In the data collection phase, an important feature is the ability to order events for visualisation. Collection of data from parallel programs differs from collection of data in sequential programs in that there are multiple streams of events. Timestamps are often used as a means of ordering [Lamp 1978].

Once the data collection phase has produced a stream of events, an analysis phase may then process these events. This processing includes the ordering of events and the detection of higher-level abstract events from the stream of lower-level events. The events collected may be low-level events such as cache miss, a slightly higher-level event such as a procedure or function call, or a higher-level user defined interesting event. In the case of a user defined event, some of the analysis has been performed in advance by the user. The determination of what to visualise must occur at some point.

The storage requirements depend in large part on both the purpose of the visualisation itself and the intended future uses of the stored information.

Many facets of the display task are determined by the purpose of the visualisation. Program visualisation systems provide highly application specific views of the program's data structures and the operations which update these data structures.

Examples of tools that perform parallel program visualisation are described in [Bode 1993] and [Joyce 1987].

4.2 Open Distributed Software Systems

We first give an overview of visualisation tools for distributed system as they are described in literature. Then we give a comparison between those tools.

4.2.1 An overview

Contrary to visualisation of parallel systems, only little research has been done concerning the visualisation of open distributed software systems. In this section we describe the efforts of Bond [Bond 1994], Brühan [Brüh 1996] and Brunne [Brun 1996]. In the following section we compare the approaches described in this section.

Bond describes in [Bond 1994] two tools for visualising service interaction in a Distributed Computing Environment (DCE). DCE is a distributed environment that supports the client/server paradigm. Interaction between objects in DCE takes place through Remote Procedure Calls (RPCs) according to the standards of the Open Software Foundation (OSF). In contrary to CORBA, DCE is not an object oriented environment.

The design of Bond's visualisation system draws upon two area's of research:
1. Event logging
2. Event visualisation

This is similar to the approach of parallel systems, as described in the previous section. Bond gives a few reasons for providing the event logging service. The most important include the provision of fault tolerance in distributed computing, as described in [Dan 1991] and the improvement of performance in parallel and distributed systems, as described in [Wyb 1988] and [Zer 1991]. The interest of Bond is in logging both the state changes in distributed objects and interaction between objects. The motivation of Bond for visualising events is that component interactions become explicit and the *larger picture* becomes clear.

Bond has developed a service for event logging called ELVIN and an animation tool called WALTER. In ELVIN the following actions are logged: object creation, state change, message passing and object destruction. The programmer has to insert event logging calls in both the client and server code. The animation tool WALTER is used to display a representation of the DCE objects generating a stream of events. WALTER provides visualisation of two types of events: a change in an object's internal state and an interaction between two objects. Initialisation and termination events are also interpreted. Objects are default represented by squares. It is possible to change the visual appearance of objects with a configuration file. Objects may be moved once displayed. The message transmission is illustrated by drawing a line between objects. The problem of ordering events is ignored.

The emphasis of Brühan in [Brüh 1996] is on monitoring CORBA applications. A programmer of location transparent systems often doesn't know whether the right remote objects are accessed. Furthermore the developer of the system knows how the components in the system interact, but for outsiders it's often not so clear what is happening inside the system. That is why the authors have developed ObjectMonitor, a system for monitoring CORBA applications. ObjectMonitor gives an Application Programming Interface (API) that can be used to trace the dynamic behaviour of the system. The output of ObjectMonitor can be used as input for a visualisation system. The

authors of the article use VISCO, which is a visualisation tool which allows to animate the dynamic behaviour of an application [Rup 1996]. With ObjectMonitor the requests to CORBA objects can be traced. To accomplish this, the authors use the filter mechanism of Orbix (Orbix is a full implementation of CORBA). The approach of Brühan is also similar to the parallel system approach. Data is collected, analysed and stored by ObjectMonitor and displayed by VISCO.

ObjectMonitor consists of three parts:
1. a client filter
2. the monitor
3. an agent for each host in the system

The client filter catches all requests that are going to a server and sends some extra data along with the request. This extra data consists of the name of the host, the name of the process, the process id and the name of the object (marker). This data is un-marshalled on the server side. In this way the server knows exactly which client has made the request. The task of the monitor is to collect the messages that are important for the visualisation and store them in shared memory. When a server on its turn acts as a client, the collected messages are sent along with the request to the server this server is communicating with. The agent is an isolated process that runs on each host in the system. The task of this agent is to make a shared memory segment on each host in which the collected messages of the local monitor objects are stored. The agent sends this information over a TCP/IP Socket Connection to a visualisation tool, in this case VISCO.

Brunne describes in [Brun 1996] the design and implementation of a management system that enables the monitoring of CORBA based applications. In this approach, CORBA based resources are to be instrumented (i.e. additional code is added to the program) in order to enable the monitoring of CORBA based applications. The ambition of the authors is to give a common and integrated view upon all resources of a distributed system, both on the network, (operating) system, middleware and application level.

The approach taken is to apply the concepts of network management to the world of CORBA based resources. Two basic principles are considered: the concept of a *managed object* and the *management protocols*. A managed object is a functional extension of an object. It mostly consists of attributes that represent properties of the component, management operations and notifications that represent asynchronous events to be reported by the managed object. There are a lot of management protocols, like the Internet approach based on the Simple Network Management Protocol (SNMP), the OSI Management approach based on the OSI Common Management Information Protocol (CMIP) and its related concepts, and the CORBA based approach using an Object Request Broker as a vehicle to transport management information. The one that is used by Brunne is CORBA.

The principal architecture is the same for all protocols: an agent provides a management view upon the resources that are subject to management in terms of managed objects. These are organised in a so called *Management Information Base* (MIB) that is externally accessed by a manager through management protocol requests. In addition, an agent may report events in the form of notifications to the manager. According to the design principles presented above, the CORBA application components have to be extended for the purpose of management. In the article a table is presented with ideas of what could be useful management information for CORBA application components. A Management Information Base (MIB) must be unambiguously defined in a formal notation in order to be accessed from any manager system that knows about the schema of the MIB. The notation the authors have chosen for this is CORBA IDL.

The programmer of the CORBA application has to add management extensions to the system in the following phases:
- Object Definition Phase
- IDL Compilation Phase
- Object Implementation Phase
- Linking Phase

- Starting Phase

The authors have implemented a management system, using Orbix. They use the Orbix filters for tracing the requests in the system. The CORBA managed objects create events, which are pushed to a CORBA event channel. These events are visualised using a NeXTStep/Openstep based visualisation tool.

4.2.2 A comparison

To conclude we give a comparison between the systems for visualisation of open distributed software systems. Table 4-1 gives an overview of the advantages and disadvantages of the various tools.

System	Advantages	Disadvantages
WALTER and ELVIN [Bond 1994]	• Separation of concerns: the event logging part and the visualisation part are separated.	• DCE is used, which is not an object oriented environment. • The event logging calls have to be added to the client and server code by the programmer of the distributed system.
ObjectMonitor [Brüh 1996]	• A CORBA system is monitored.	• A complicated method is used to retrieve information from the system (no separation of concerns). • The programmer of the distributed system has to add code to the client and server code.
Management system [Brun 1996]	• A CORBA system is monitored. • Separation of concerns: the requests are filtered and generate events, which are handled to produce the visualisation.	• The programmer has to add *management information* during the implementation phase of the distributed system. • The programmer has to add a lot of extra code in all stages of the implementation phase.

Table 4-1: Comparison between visualisation tools for distributed systems

The tools that are described in this section have too many disadvantages to be useful for us. Therefore we have developed our own visualisation tool without the disadvantages as they are described in Table 4-1. In the following chapter we describe the design and implementation issues of our tool.

5. Visualisation in an Open Distributed Environment

This Chapter first describes a generic method for visualising open distributed software systems. ODP-RM is used to define what kind of information can be extracted from a distributed system, but also for describing the information that has to be visualised. We have applied this method in the design of a tool that visualises CORBA systems. This is described in the second section. The third section describes the implementation of the tool. The Chapter concludes with an evaluation of the current version of the tool.

5.1 A Generic Method for Visualising Open Distributed Systems

When visualising open distributed software systems, a process of four phases has to be performed:

1. The *event collection* phase, during which the available information is retrieved from the system.
2. The *event processing* phase, during which the collected events are ordered and translated into events that are used in the visualisation.
3. The *storage* phase, during which the visualisation events are stored in a queue, or a log file.
4. The *display* phase, during which the visualisation events are translated into a graphical representation (i.e. shape, colour, position, etc.) and displayed.

An important part of the visualisation process is to create a set of events that gives a full description of the actions that are performed in the system that is visualised. We distinguish two types of events:

- *ORB events*: events that are retrieved during the event collection phase and are input for the event processing phase.
- *Visualisation events*: events that are displayed during the display phase. These events are the output of the event processing and storage phase.

Figure 5-1 shows the in- and output of each of the four phases in the visualisation process.

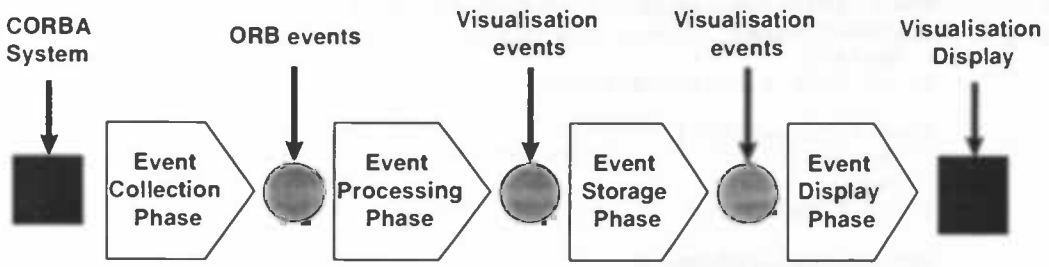


Figure 5-1: The visualisation process

In the following paragraphs we give a description of the ORB and Visualisation events along with their IDL specification.

The ORB events should collect all the available information from the distributed system that is visualised. Table 2-3 gives a full description of the information that can be obtained from the system. We have constructed three types of ORB events that can be regarded as *subclasses* from the *structure ORBaction* that describe all kinds of actions that can happen in a distributed system. Figure 5-2 gives in UML notation the structure of the ORB events.

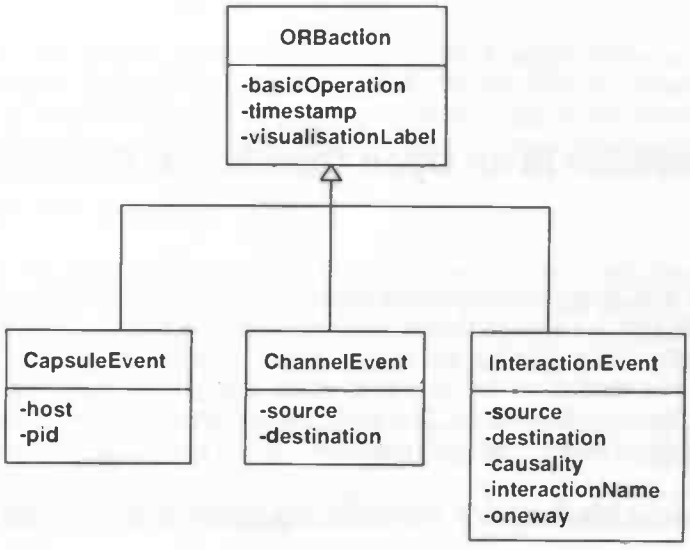


Figure 5-2: : Schematic overview of ORB events

Every ORB action has three basic attributes:

- `basicOperation`: every action has an `instantiate` or a `destroy` tag.
- `timestamp`: the time an event happens.
- `visualisationLabel`: a label that is used by the graphical user interface to give a certain object a pre-defined appearance. If this label is not set, the object gets the standard appearance.

In addition to these attributes, the `CapsuleEvent` has a `host` and a `pid` attribute that give a unique identification for the capsule. This kind of event is generated when a capsule is created or destroyed. The `ChannelEvent` has attributes to determine what the `source` and `destination` of the channel is. This kind of event is generated when a channel is created or destroyed. The `InteractionEvent` has attributes for the `source` and `destination` of the interaction, the `causality` (i.e. whether the operation is a request or a reply), the name of the interaction (`interactionName`) and an attribute to determine whether the interaction is a `oneway` request. The IDL specification of ORB events is given below. Because OMG-IDL doesn't support inheritance of structures, the structures of Figure 5-2 are flattened to be able to describe them in IDL.

```
Enum t_BasicOperation {
  instantiate,
  destroy
}; // Enum t_BasicOperation

Enum t_Causality {
  request,
  reply
}; // t_Causality

Struct CapsuleEvent {
  t_BasicOperation basicOperation;
  string timestamp;
  string visualisationLabel;
  string pid;
  string host;
}; // struct CapsuleEvent

Struct ChannelEvent {
  t_BasicOperation basicOperation;
  string timestamp;
  string visualisationLabel;
  string pidFrom;
  string hostFrom;
  string pidTo;
```

```
  string hostTo;
); // struct ChannelEvent

Struct InteractionEvent {
  t_BasicOperation basicOperation;
  string timestamp;
  string visualisationLabel;
  string pidFrom;
  string hostFrom;
  string pidTo;
  string hostTo;
  t_Causality causality;
  boolean oneway;
  string operationName;
); // struct InteractionEvent
```

The set of Visualisation events is larger than the set of ORB events. It must be possible to generate events for every viewpoint of ODP-RM. The set that we have created is not yet complete, and will be extended in the future, based on the experiences of users of our tool. All the ORB events are also Visualisation events. At this moment we have designed two extra Visualisation events that can be generated (see Figure 5-3).

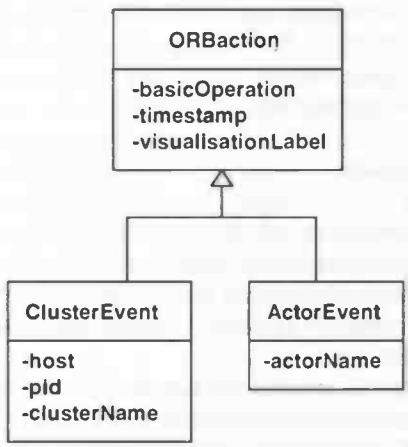


Figure 5-3: Schematic overview of a part of the Visualisation events

Like the ORB events, the `ClusterEvent` and the `ActorEvent` inherit all the attributes from the structure `ORBaction`. The `ClusterEvent` has attributes to determine the identity of the capsule that the cluster belongs to (`host` and `pid`). The `marker` gives a unique name for the cluster within the capsule. The `ActorEvent` has an attribute that gives the `name` of the actor. The IDL specification of those Visualisation events is given below.

```
struct ClusterEvent {
  t_BasicOperation basicOperation;
  string timestamp;
  string visualisationLabel;
  string host;
  string pid;
  string clusterName;
} // ClusterEvent

struct ActorEvent {
  t_BasicOperation basicOperation;
  string timestamp;
  string visualisationLabel;
  string actorName;
} // ActorEvent
```

5.2 Design of a Visualisation Tool

In this section the design of a tool for visualising CORBA applications is described. The tool is called OBVIouS (OBject VIsualisation System). In the first paragraph we determine the target group of users of the tool. Then we give an analysis of the requirements for the tool. The last paragraph describes the architecture that we have designed for our tool.

5.2.1 Target group of users

At KPN Research Program Visualisation is used to gain a better understanding about the behaviour of a program in a distributed environment. Distributed programs can be very complex. Even more complex can be to explain to other people how such systems work. A good visualisation tool would be very useful in this case. The purpose of the visualisation depends on the target group of users:

- Researchers and developers of distributed systems could use program visualisation for debugging purposes and for gaining insight in the information flow of the program. With the tool they could also detect potential bottlenecks in the communication.
- Managers could use program visualisation to get a high-level understanding of the distributed system.
- System designers and architects could use the tool to see if the CORBA system has been implemented according to their design.
- End users of the distributed system could use program visualisation to get an idea of the things that happen in the system.

5.2.2 Requirements analysis

From interviews with developers of CORBA systems at KPN Research, we have composed a set of system requirements, which apply to our tool. This set is listed below.

- The efforts on the software developer for visualising an application should be minimal. This implies that the developer shouldn't have to add much code to the application for performing the visualisation.
- The tool must be capable of visualising multiple views on the same system. We define these views using the viewpoints from the Open Distributed Processing - Reference Model (ODP-RM). This makes the tool applicable to many target groups of users.
- The output format of the tool must be easy to configure. This means that it must be easy to change the look of the objects that are visualised.
- Online visualisation must be possible, i.e. the visualisation is performed while the distributed system is running.
- It should be possible to run the visualisation both in *step mode* and *continuous mode*.
- Detailed information must be displayed when demanded (e.g. the computing system an object is running on).
- It should be possible to make templates of what you want to see during a visualisation session. It must be possible to switch runtime between the templates.

We have tried to meet al those requirements in our tool.

Most of the wishes of the developers can be modelled with use cases. The notation of the Unified Modelling Language (UML) is used, which is described in appendix A. Figure 5-4 shows a use case of the tool for one developer of CORBA systems. It gives an idea of the interaction of a user with the tool.

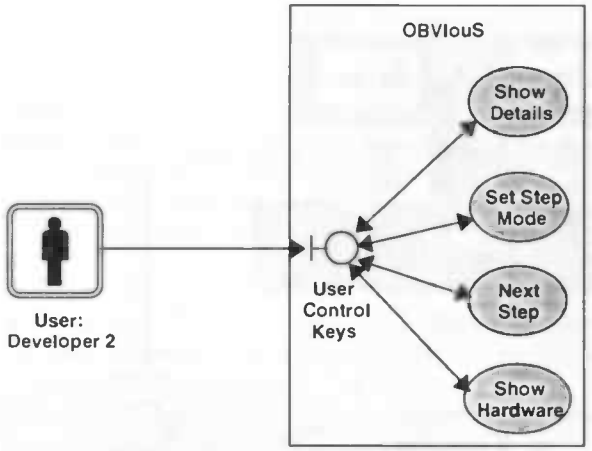


Figure 5-4: Use Case for OBVIouS

Besides the wishes of the developers, there are a few more restrictions that have to be considered. These restrictions are based on the study of literature in Chapter 4. From the advantages and disadvantages of the systems that were described in that Chapter (see Table 4-1), we constructed a second set of requirements. These requirements partially overlap the requirements of the developers.

- The architecture of our visualisation system should have a good separation of concerns. The event logging part and the visualisation part should be separated, like the systems described in [Bond 1994] and [Brun 1996].
- The programmer of the CORBA system shouldn't have to add much code, unlike all the systems described in Section 4.2 (see Table 4-1).
- The programmer of the CORBA system has to add design information to the system, but not during the implementation phase of the system that is visualised. This is contrary to the visualisation system as it was described in [Brun 1996].

The following section describes the design and implementation of the tool that we have developed to accomplish both the requirements derived from literature and the requirements of the developers of the CORBA systems.

5.2.3 Tool Architecture

For the design of a good architecture, we used the method of Section 5.1. The visualisation process that is described in that Section consists of four phases:

- Event Collection Phase
- Event Processing Phase
- Storage Phase
- Display Phase

We have designed an architecture that handles these phases separately. This architecture is shown in Figure 5-5.

The generation of ORB events is performed in *Filters*, that push the events into an *EventChannel*, for which we use the CORBA Event Service. In this phase of the process the question is answered *which actions are performed by the CORBA system.*

The EventChannel pushes on its turn the ORB events into a server that performs the event processing. The ORB events are translated into Visualisation events by the *Rule Engine* of this server. In this phase it has to be determined *what has to be visualised.*

The processed events are stored in an event queue and a log file during the third phase.

We have designed a graphical user interface that retrieves the visualisation events from the event queue, adds display information for the visualisation and displays the information. In this phase an important question is *what appearance should the events have.*

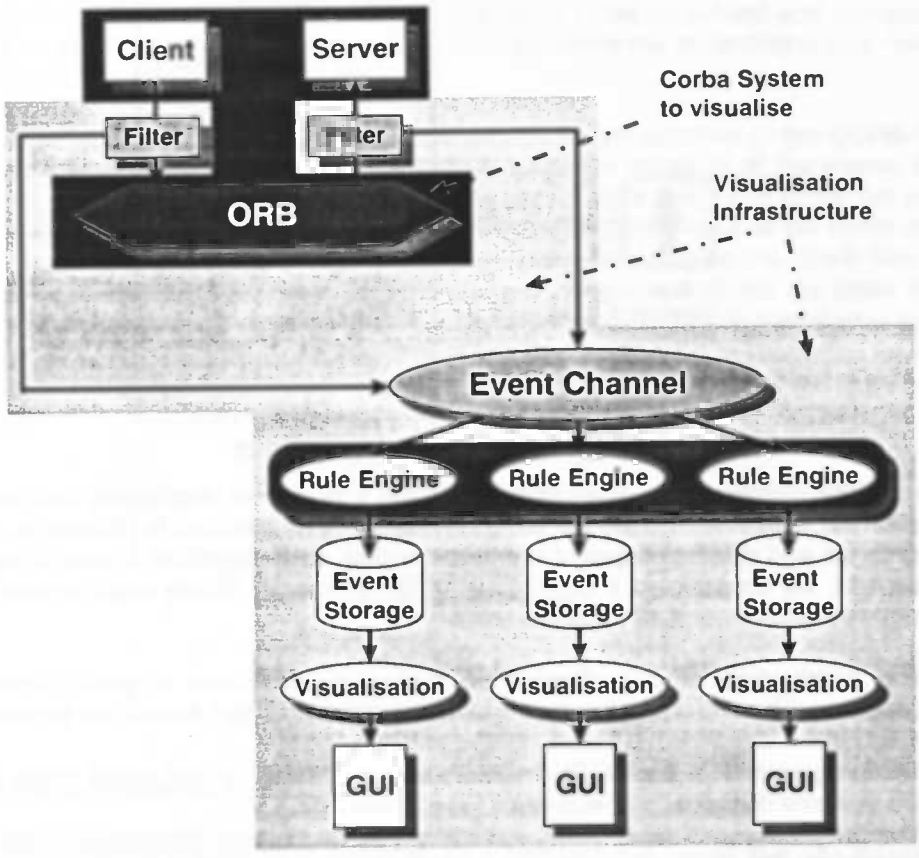


Figure 5-5: Architecture of the OBject VIsualisation System (OBVIouS)

5.3 Implementation of the Tool

This section describes the implementation issues of the tool. The first Paragraph describes the Event Collection phase. The second Paragraph describes the Event Processing and Storage phase. The last paragraph describes the implementation of the Event Display phase.

5.3.1 Event Collection

Many of the distributed systems at KPN Research have been built with Orbix, a full implementation of CORBA, from Iona Technologies Ltd. That is the reason why we have chosen to build our visualisation tool with Orbix too.

The ORB events that we want to collect from the system that has to be visualised have to be generated at some point in that system. Orbix has a filter mechanism which gives the possibility to add code at certain *filter points* to retrieve information about a certain request. Orbix provides ten filter points, of which eight marshalling points and two failure points. There are marshalling points available for filtering *pre* and *post* marshal. We use the four *post* marshal points, because they give information about parameters of an operation that is performed contrary to the pre marshal points (see Figure 5-6):

1. *out request post marshal*: [in the caller's address space] before an operation or attribute request is transmitted from the filter's address space to any object in another address space; in particular, after the operation's parameters have been added to the request packet.
2. *in request post marshal*: [in the target object's address space] once an operation or attribute request has arrived at the filter's address space, but before it has been processed; in particular, before an operation has been sent to the target object and after the operation's parameters have been removed from the request packet.

3. *out reply post marshal*: [in the target object's address space] after the operation or attribute request has been processed by the target object, but before the result has been transmitted to the caller's address space; in particular, after an operation's out parameters and return value have been added to the reply packet.
4. *in reply post marshal*: [in the caller's address space] after the result of an operation or attribute request has arrived at the filter's address space, but before the result has been processed; in particular, after an operation's return parameters and return value have been removed from the reply packet.

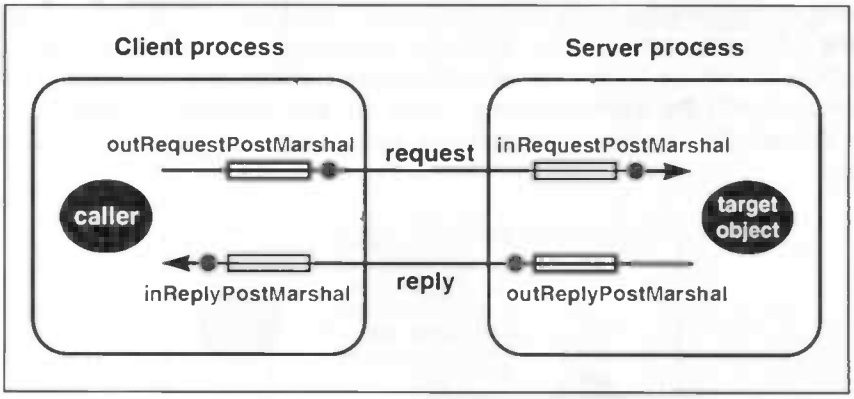


Figure 5-6: Filter monitor points

We have built C++ filter objects that use the filter mechanism. The information that passes these filter objects gives all the information we need to generate all types of ORB events. When the filter objects are instantiated or deleted, we know that a capsule has been created or destroyed. When a channel is constructed or deleted, we let Orbix notify the filters about it. In this way we know exactly when to generate a channel event. The filter points in the filter object provide information about requests that are made between a client and a server. We use the filter points to trace in-requests, out-requests, in-replies and out-replies. When a certain request/reply is made, we generate an interaction event.

Figure 5-7 shows the class diagrams of the filter objects in UML notation. For servers we have constructed the `ServerFilter` and for clients the `ClientFilter`. If a server is also a client, he has a filter that is a combination of the `ClientFilter` and the `ServerFilter`. The `dataPS`, `eventChannelName`, and `otalk` attributes are used to make a connection with the event channel. The server has an extra attribute `clientId` for the identity of the client that is send along with the request from a client to a server. The `outRequestPostMarshal`, `inRequestPostMarshal`, `outReplyPostMarshal`, and the `inReplyPostMarshal` methods are the filtering points that we described above. The `ClientFilter` and the `ServerFilter` class are subclasses from the class `CORBA::Filter`.

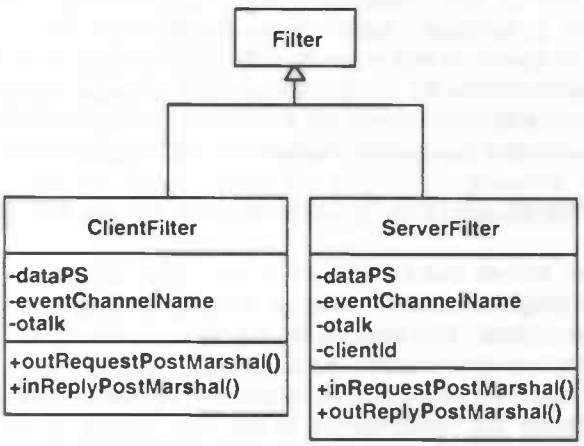


Figure 5-7: Client- and ServerFilter classes

To use the filter objects the developer only has to add three lines of code to the main programs of the client and server objects that have to be visualised and compile those programs again. The code that has to be added performs the instantiation and deletion of the filter objects.

For the generation of events we use OrbixTalk, an implementation of the CORBA Event Service of Iona Technologies Ltd. OrbixTalk handles the transport of the events through the event channel to interested server objects (see Figure 5-5). The filters act as *push suppliers* for the event channel. This means that they supply events to the event channel. We have implemented a `DataPushSupplier` class that handles this. Each filter object contains a `DataPushSupplier` attribute `dataPS`. Figure 5-8 shows the class diagram of the `DataPushSupplier` class in UML notation. The class has four attributes to set up a connection with the event channel. There are two operations available: `attach()` to attach the supplier to the event channel and `pushData()` to push the data into the channel.

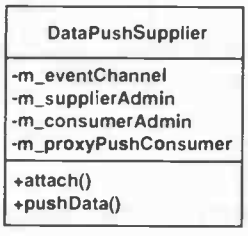


Figure 5-8: DataPushSupplier class

5.3.2 Event Processing and Storage

To process the events, we have designed an Event Listener server that can translate the ORB events into Visualisation events. However, this translation depends on the design of the CORBA system and on the kind of view that has to be created. Because we want the visualisation to be generic, the translation should not be programmed in the Event Listener server itself, but in a script language. Without recompilation of the Event Listener server we can take another script that performs another translation of the events. The script serves as a *rule engine* in the visualisation process (see Figure 5-5). In our implementation we use Perl as the script language. Perl is very good at pattern matching and manipulation of strings. We offer the events to the Perl script as strings, so the translation can be programmed quite efficiently. An example of a Perl script is given in Appendix C.

The ORB events that are pushed to the Event Listener server are stored in a thread safe queue. An event thread processes these events by pushing them into the Perl script and putting the result that is returned in another queue. A problem here is the ordering of the ORB events. The events don't necessarily arrive at the Event Listener server in the same order as they were generated. Therefore the events that come in at the Event Listener server have to be ordered. At this moment it is only possible to make a partial ordering on the interaction events (requests are performed in the order: instantiate-request, destroy-request, instantiate-reply, destroy-reply). For the absolute ordering we need to use the timestamps that we send along with the events. The partial ordering is performed in the Perl-script, but the absolute ordering is not implemented yet. We assumed that the partial ordering gives at this moment enough information to provide a good view on the system.

The Event Listener server has been built in C++ with Orbix 2.2. We have designed an architecture containing three computational objects: a factory object, an administration object and a core object. The factory is the capsule manager of the server and the administration object is the cluster manager (see Table 2-3). These objects provide interfaces for instantiation and deletion of server objects and computational objects. All user-defined interfaces are implemented in the core object. In the Event Listener server we have two interfaces (see Figure 5-9):

- The `PushConsumer` interface which implements the connection with the event-channel. This interface provides the method `push(any)` which is implemented in the `ELCore` object. The IDL specification of this interface is [OMG-2 1997]:

```
interface PushConsumer {
  void push (in any data) raises(Disconnected);
  void disconnect_push_consumer();
};
```

- The `EventListener_IF` interface through which clients can retrieve the next Visualisation event. The method that is provided is `getNextEvent()`, which is implemented in the `ELCore` object. The IDL specification of this interface is:

```
interface EventListener_IF {
  VisualisationEvent getNextEvent();
};
```

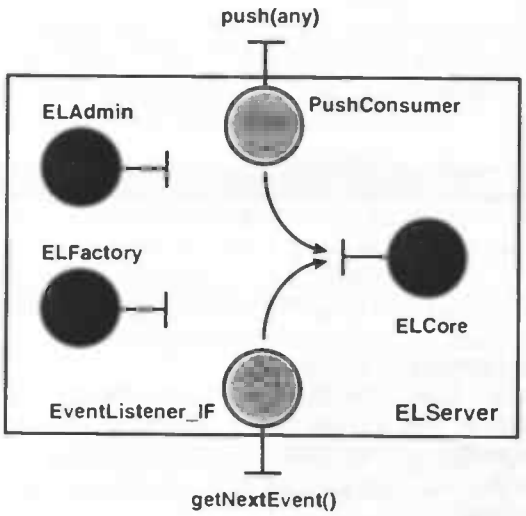


Figure 5-9: Architecture of the Event Listener server (ELServer)

To make different views on the system the developer of the CORBA system has to write a Perl script for every view. To run the different views simultaneously different instances of the Event Listener should be created with a unique name, that is called a *marker*. The name of the marker must be the same as the name of the Perl script, without the extension *pl*. These instances can be created with the user interface that we have designed. This user interface is explained in more detail in the following section.

5.3.3 Display of the Events

For the display of Visualisation events we have designed a graphical user interface in Java. This user interface consists of three parts:

- A control panel, on which can be specified what the name of the view is, what the name of the host is on which the Event Listener server runs and whether the events have to be collected manually or automatically. Furthermore, the control panel has buttons to connect to and disconnect from the Event Listener server.
- A feedback area, on which textual information is displayed about what is happening in the system.
- A screen on which the events that are retrieved from the Event Listener server are displayed.

Before events can be displayed on the screen, it has to be determined what the appearance of the events is. For this we have constructed a configuration file, which gives the possibility to add this kind of information to the user interface. For each view on the CORBA system a new configuration file has to be created.

The configuration file must be written according a specified grammar. For this grammar we have constructed a scanner and a parser using the scanner generator JLex [Berk 1997] and the parser generator CUP [Hudson 1996]. This gives the possibility to easily

adapt the configuration grammar to new requirements. The grammar that is currently in use is shown below.

```
Config           ::=  viewName width height ColorRule
                      Capsules InteractionRule ChannelRule |
                      viewName width height 'file' fileName
                      Capsules InteractionRule ChannelRule ;

Capsules         ::=  Capsules CapsuleRule |
                      CapsuleRule ;

CapsuleRule      ::=  'capsule' x y visualisationLabel
                      ObjectShape |
                      'capsule' ALIGNMENT visualisationLabel
                      ObjectShape ;

InteractionRule  ::=  'interaction' ColorRule nrInteractions;

ChannelRule      ::=  'channel' ColorRule ;

ObjectShape      ::=  'rectangle' width height ColorRule |
                      'circle' radius ColorRule |
                      'file' fileName ;

ColorRule        ::=  COLOR |
                      'rgb' red green blue ;
```

The configuration file consists of four parts:

1. General information. The name of the view must be specified along with the width and height of the display. The display has a minimum size that is used if the width and height that are specified are too small. In this part of the configuration the background also has to be specified. This can either be a colour or an image that is present in a file (in the GIF-format).
2. Information about the capsules that are displayed. For every capsule a configuration may be specified. Every capsule has become a unique visualisation label in the filter that is used here (see Section 5.1). The position of the capsule must be specified, either in exact co-ordinates or with an area on the display (e.g. south, north-east, middle, etc.). Furthermore, the shape of the capsule must be given. This can be a rectangle, circle, or a shape that is read from a file (in GIF-format). If no configuration is given, the capsule gets a default appearance that may also be specified in the configuration. The `visualisationLabel` for the default configuration of capsules is `default`.
3. Information about interactions. For an interaction the colour must be specified along with the maximum number of interactions that are displayed on the screen at the same time between two capsules.
4. Information about channels. At this moment it is only possible to set the colour of the channel.

An example of a configuration file is shown in Appendix D.

In Section 3.2 we studied the principles of making good visual representations. We have tried to apply those principles to the graphical user interface that we have designed.

A visualisation of an open distributed software system is a graph. In Paragraph 3.2.1 we discussed the principles of drawing graphs. These principles cannot be used here, because the graph that we have is too dynamic, and when the graph is extended, we don't want to change the place of the objects that are already on the display. We have found a workable solution for this problem. In the configuration file the place of objects can be specified. During the visualisation this place can be changed by *dragging* the object to another place on the display.

The colours that are used in the visualisation are specified in the configuration file. Therefore it's the responsibility of the creator of this file to select good colours. The

colours in the graphical user interface that cannot be changed, are the colours of the buttons and the background-colour of the applet. We have selected light yellow and light grey for these items, to make them not too dominant.

The animation principles from Paragraph 3.2.3 have not been applied yet.

For the design of the components in the graphical user interface we have applied the ideas of Paragraph 3.2.4. We have tried to make a good composition of the components in the window and use appropriate labels and good pointers.

A screendump of the graphical user interface is shown in Appendix B.

5.4 Evaluation of the Tool

We organised a demonstration day to show future users the possibilities with the tool that we have developed so far. Most of these users intend to apply our tool for visualising their CORBA systems. They gave us feedback on how to extend the tool to make it useful in their projects. Table 5-1 summarises their recommendations.

Recommendation	Possible Solution
There should be a legacy connection. This implies that it should be possible to use the tool for other than CORBA systems.	The concerned legacy environment must have filters that generate the appropriate events.
It should be possible to save new locations of objects in the configuration file.	The user interface must be adapted with an option to save the new configuration. The implementation of this is not very easy because of security problems in Java.
It should be possible to print a trace diagram of the events that happened during the visualisation.	Every change in the state of the visualisation must be saved in an appropriate format (e.g. Windows Metafile). In this way it can be used in programs like Word and PowerPoint. An other possibility is to make a connection with programs like Scenar that are good at processing trace diagrams.
It should be possible to visualise Computational Objects. This means that it has to be visible on which interfaces which interactions take place.	At this moment a CorbaObjectEvent is generated when an interface starts to be active. This information could be used in the Rule Engine to generate the ComputationalObjectEvent, which is already available. However, the user interface must be adapted to visualise those events.
It should be possible to see the time on which the event was generated.	The events already have an attribute in which the time can be stored. When these time stamps are used, it has to be considered that the time on different machines need not be the same. This means that an event that was generated earlier can have a later time stamp than an event that was generated later.

It should be possible to start the visualisation tool later than the system that is visualised without losing any events that have been generated so far.	A message store should be used by the event channel to store the events that have been generated.
It should be possible to group objects in the display (runtime).	This will have impact on the user interface. Data structures are needed to save objects that belong to a group. Furthermore, it must be possible to select objects from the display and group them. This has also impact on the interactions. Interactions that are in a group may not be displayed anymore contrary to interactions between groups.
It should be possible to use the tool in combination with other ORB's than Orbix.	The other ORB's need to generate the ORB events. If they have a filter construction like Orbix, this can be done quite easy. Otherwise the code for generation of the events has to be added with every call that is made.
It should be possible to use one graphical user interface to look at the different views.	For this we need to implement tab-pages in Java.
It should be possible to print the number of interactions instead of an arrow when an interaction takes place.	This recommendation means that it should be possible to view statistic information in the system. For this Statistic events should be generated in the Rule Engine instead of Visualisation events. These events should be processed by another kind of user interface. Furthermore, couplings could be made with statistical applications.

Table 5-1: Recommendations for extension of the tool's possibilities

6. Conclusions and Recommendations

This Chapter presents the conclusions of this thesis. It gives recommendations for extending the tool. Furthermore it gives some recommendations towards ORB vendors for extending their ORB's.

6.1 Conclusions

Developing distributed systems is a complex task. Even more complex is to explain to other people how a particular distributed system works. Until recently such systems could only be demonstrated by means of scrolling text windows accompanied by pictures that explained the architecture of the system. Program visualisation can help a lot in gaining more insight in how such systems work, more than scrolling text and static pictures.

In this thesis we answer the question of *how to visualise applications in open distributed environments*. In our approach we developed a generic method for visualising distributed applications and applied this method to CORBA systems.

The method describes a visualisation process in four phases: the event collection, event processing, storage and display phases. These phases have proven to be necessary, in order to allow a large degree of flexibility in what is visualised and the appearance of visualisation entities. Concepts of ODP-RM have been analysed and a subset has been chosen to identify concepts suitable for visualisation. In understanding the key features of the distributed system, ODP-RM provides a good reference. We have defined a mapping from entities in a CORBA system to ODP-RM concepts. This gives us the possibility to describe the distributed system with ODP-RM and then translate this description into events that occur in the distributed system.

With this generic method we constructed a highly configurable tool, called OBVIouS, that can visualise CORBA systems while they are operating. Our tool gives the possibility to produce different views on the system simultaneously. The programming efforts for the developer of the CORBA system are kept low. Only one rule has to be added to the main program of every client- or server process to enable visualisation. This rule performs the instantiation of so-called Filter objects that generate the ORB events. We have applied the CORBA Event Service for receiving and distributing the events. This service allows a loose coupling of the visualisation tool from the system that has to be visualised. By means of a Perl script the events generated from the CORBA system, can be filtered out or transformed into visualisation events. A configuration file is used to instruct a Java applet on the appearance and representation of visualisation events.

6.2 Recommendations

Currently, our visualisation tool has been implemented with Orbix. In our tool we use the Orbix specific filter mechanism, which gives us the possibility to generate events when requests are made. It should be possible to generate the same events from other ORB implementations if they provide hooks for intercepting requests. We recommend that ORB implementations provide those hooks to make it more easy to integrate the visualisation system with the application that is visualised.

Furthermore we recommend that ORB implementations provide a standard set of events that can be generated in a CORBA system. This implies that an application developer doesn't have to add any code to make a visualisation possible. The ODP-RM provides a

nice set of concepts that can be used to define a standard set of visualisation events. This set should be standardised by the OMG.

In addition, the ORB could provide a management interface to configure the level of detail of the visualisation events. This implies that during the performance of the visualisation tool, it can be decided that some of the events don't have to be generated, because they are never used. This improves the performance of the system.

The tool that we have designed is more suitable for visualising interactions in a CORBA system than the tools that were described in Paragraph 4.2.2. However, the tool should be extended to make it more useful. Table 5-1 gives a summary of recommendations for future work from the user's point of view. We recommend that this feedback is taken into account in future developments on OBVIouS.

Literature References

[Berk 1997] E. Berk, '*Jlex: A lexical analyser generator for Java ™*', Department of Computer Science, Princeton University, http://www.cs.princeton.edu/~appel/modern/java/Jlex/manual.html, valid in December 1997, 1997.

[Bode 1993] A. Bode and P. Braun, *'Monitoring and Visualisation in TOPSYS'*, Performance Measurement and Visualisation of Parallel Systems, Proceedings of the workshop, pp 97-118, Elsevier Science Publishers B.V., 1993.

[Bond 1994] A. Bond and D. Arnold, *'Visualising Service Interaction in an Open Distributed System'*, proceedings of IEEE Workshop on Services for Distributed and Networked Environments, pp 19-25, 1994.

[Bran 1988] F.J. Brandenburg, *'Nice Drawings of Graphs are Computationally Hard'*, Lecture Notes in Computer Science 439, pp 1-15, Springer-Verlag, 1988.

[Brown 1985] M.H. Brown and R. Sedgewick, *Techniques for algorithm animation'*, IEEE Software 2(1), pp 28-39, 1985.

[Brown 1991] M.H. Brown and J. Hershberger, *'Colour and Sound in Algorithm Animation'*, DEC SRC Report 76a, ftp://gatekeeper.dec.com/pub/DEC/SRC/research-reports/SRC-076a.ps.Z, valid in November 1997, 1991.

[Brüh 1996] M. Brühan and S. Ruppert, *'ObjectMonitor: Monitoring von CORBA-Anwendungen'*, Fachhochschule Wiesbaden, http://wwwvs.informatik.fh-wiesbaden.de/programme/objmon-visco/, valid in November 1997, 1996.

[Brun 1996] H. Brunne and T. Usländer, *'Design of a Monitoring System for CORBA-based Applications'*, Trends in Distributed Systems '96, Industrial and Short Paper Proceedings, Workshop RWTH Aachen, pp 52-66, 1996.

[Dan 1991] D. Daniels, R. Haskin, J. Reinke and W. Sawdon, *'Shared logging services for fault-tolerant distributed computing'*, ACM SIGOPS Operating Systems Review, vol. 25, pp 65-68, 1991.

[Fowler 1995] S.L. Fowler and V.R. Stanwick, *'The Gui Style Guide'*, Academic Press Limited, 1995.

[Hudson 1996] S.E. Hudson, *'LALR Parser Generator for Java ™ Cup'*, Graphics Visualisation and Usability Centre, Georgia Institute of Technology, http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/manual.v0.9e.html, valid in December 1997, 1996.

[Joyce 1987] J. Joyce, G. Lomow, K. Slind and B. Unger, *'Monitoring Distributed Systems'*, ACM Transactions on Computer Systems, vol. 5, no. 2, pp 121-150, 1987.

[Krae 1993] E. Kraemer and J.T. Stasko, *'The visualisation of Parallel Systems: An Overview'*, Journal of Parallel and Distributed Computing, vol. 18, no. 2 pp 105-17, 1993.

[Lamp 1978] L. Lamport, *'Time, clocks and the ordering of events in a distributed system'*, Communications of the ACM, vol. 21, no. 7, pp 558-565, 1978.

[Leyd 1997] P. Leydekkers, *'Multimedia Services in Open Distributed Telecommunications Environments'*, Ph.D. thesis KPN Research / University of Twente, 1997.

[Nank 1996] M.A. Nankman, *'A Reference Model for Open Distributed Storage Architectures'*, M.Sc. thesis KPN Research / State *university* of Groningen, 1996.

[ODP-1 1995] ITU/ISO, Open Distributed Processing - Reference Model, *'Part 1: Overview'*, International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[ODP-2 1995] ITU/ISO, Open Distributed Processing - Reference Model, *'Part 2: Foundations'*, International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[ODP-3 1995] ITU/ISO, Open Distributed Processing - Reference Model, *'Part 3: Architecture'*, International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[ODP-4 1995] ITU/ISO, Open Distributed Processing - Reference Model, *'Part 4: Architectural Semantics'*, International Standard 10746-3, ITU-T Recommendation X.903, 1995.

[OMG-1 1997] Object Management Group Homepage, http://www.omg.org, November 1997.

[OMG-2 1997] Object Management Group, *'The Common Object Request Broker: Architecture and Specification, Revision 2.0'*, OMG document 97.2.25, 1997.

[Orf-1 1997] R. Orfali, D. Harkey and J. Edwards, *'Instant CORBA'*, John Wiley & Sons, Inc., 1997.

[Orf-2 1997] R. Orfali and D. Harkey, *'Client/Server Programming with JAVA™ and CORBA'*, John Wiley & Sons, Inc., 1997.

[Para 1996] Paradigm Plus, *'Methods Manual'*, PLATINUM Technology inc., 1996.

[Price 1994] B.A. Price, R.M. Baeker and I.S. Small, *'A Principled Taxonomy of Software Visualisation'*, Journal of Visual Languages and Computing 4(3), pp 211-266, http://www-cs.open.ac.uk/~doc/jvlc/JVLC-Body.html, valid in November 1997, 1994.

[Rup 1996] S. Ruppert, M. Brühan and Oliver Bildesheim, *'VISCO 1.0 - VISualisation of Corba Objects'*, Fachhochschule Wiesbaden, http://wwwvs.informatik.fh-wiesbaden.de/programme/objmon-visco/, valid in November 1997, 1996.

[Tomas 1994] G. Tomas and W.C. Ueberhuber, *'Visualisation of Scientific Parallel Programs'*, Lecture Notes in Computer Science 771, Springer-Verlag, 1994.

[Tufte 1990] E.R. Tufte, *'Envisioning Information'*, pp 81-96, Graphics Press, 1990.

[UML 1997] Unified Modelling Language Homepage, http://www.rational.com/uml, valid in November 1997, 1997.

[Wyb 1988] D. Wybranietz and D. Haban, *'Monitoring and performance measuring distributed systems during operation'*, Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems, ACM Performance Evaluation Review, pp 197-206, 1988.

[Zer 1991] D. Zernik and L. Rudolph, *'Animating work and time for debugging parallel programs: Foundation and experience'*, Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN NOTICES, pp 46-56, 1991.

Abbreviations

API	-	Application Programming Interface
BEO	-	Basic Engineering Object
BOA	-	Basic Object Adapter
CMIP	-	Common Management Information Protocol
CORBA	-	Common Object Request Broker Architecture
DCE	-	Distributed Computing Environment
DII	-	Dynamic Invocation Interface
DSI	-	Dynamic Skeleton Interface
GUI	-	Graphical User Interface
IDL	-	Interface Definition Language
ODP-RM	-	Open Distributed Processing - Reference Model
IIOP	-	Internet Inter ORB Protocol
IOR	-	Interoperable Object Reference
ISO	-	International Standardisation Organisation
ITU-T	-	International Telecommunication Union
KPN	-	Koninklijke PTT Nederland
OBVIouS	-	OBject VIsualisation System
OMG	-	Object Management Group
ORB	-	Object Request Broker
OSF	-	Open Software Foundation
RPC	-	Remote Procedure Call
SNMP	-	Simple Network Management Protocol
UML	-	Unified Modelling Language

Appendix A Unified Modelling Language

The Unified Modelling Language (UML) is a common set of modelling concepts with a uniform notation. It was developed jointly by Grady Booch, Ivar Jacobson and Jim Rumbaugh at Rational Software Corporation, with contributions from other leading methodologists, software vendors and many users [UML 1997] [Para 1996].

UML provides:

- Business process modelling with use cases.
- Class and object modelling.
- Component modelling.
- Distribution and deployment modelling.

For the notation UML uses diagrams. We describe for each modelling concept the diagrams that should be designed.

A.1 Business process modelling

Business process modelling can be done with *use-case diagrams*. Use-case diagrams show the system's use-cases and which actors interact with them (see Figure A-1).

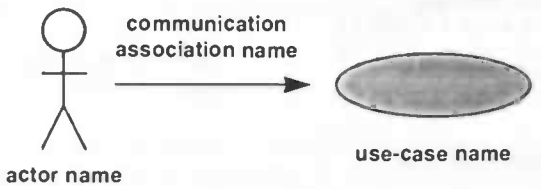


Figure A-1: Use-case diagram

A.2 Class and object modelling

For class and object modelling three types of diagrams are used: *class diagrams*, *state-transition diagrams* and *interaction diagrams*.

A class diagram shows the existence of classes and their relationships in the logical view of a system. Classes are described with their attributes and operations. Concepts like associations, generalisation and dependency are modelled in this diagram. The principles of a class diagram are shown in Figure A-2.

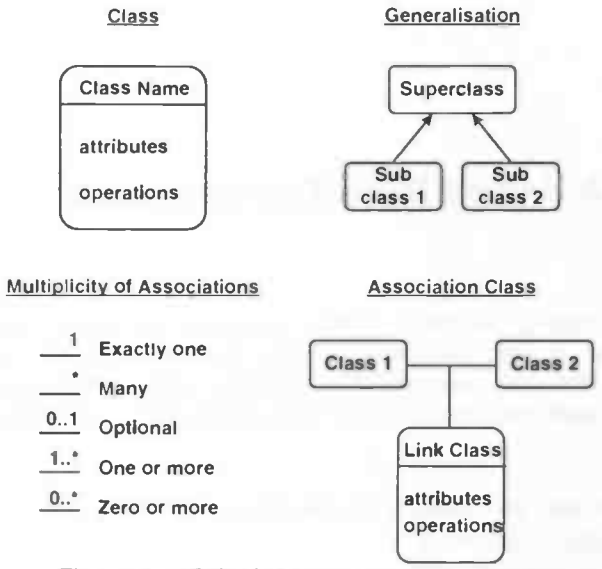


Figure A-2: Principles of a class diagram

The state-transition diagram shows the state space of a given context, the events that cause a transition from one state to another and the actions that result. Figure A-3 shows the elements that a state-transition diagram is built with.

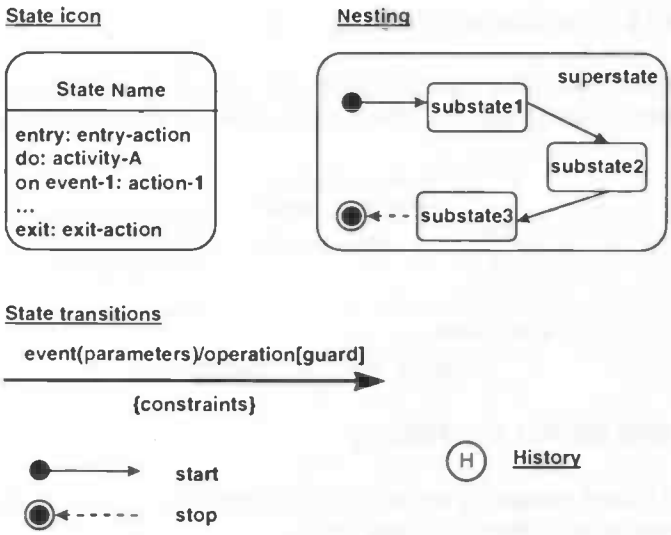


Figure A-3: State-transition elements

A interaction diagram shows the objects in the system and how they interact. There are two types of interaction diagrams: *sequence diagrams* and *collaboration diagrams*. Those diagrams are described in Figure A-4 and Figure A-5.

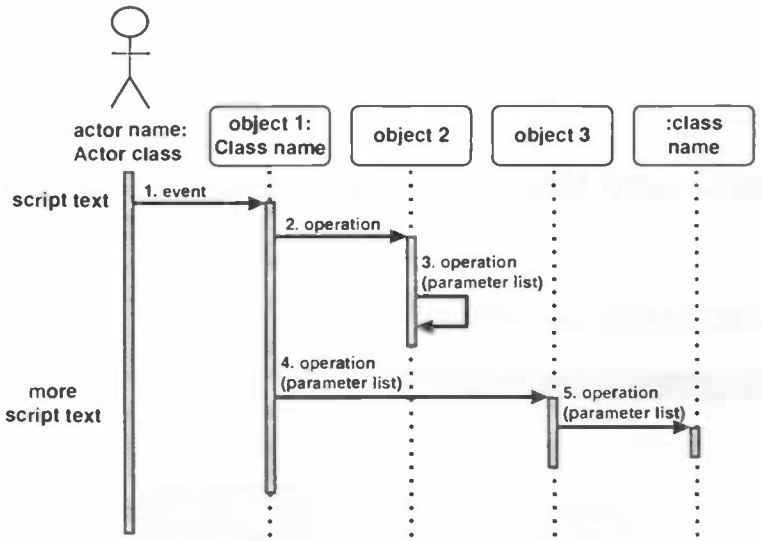


Figure A-4: Sequence diagram

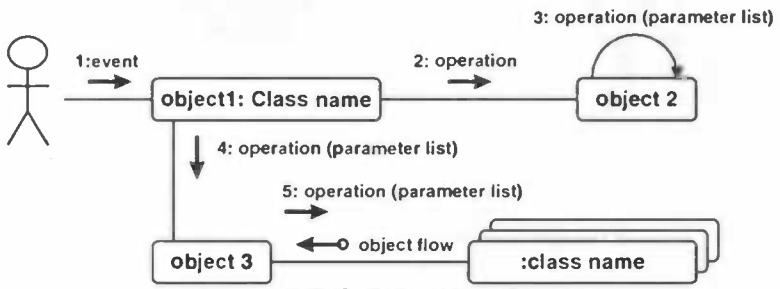


Figure A-5: Collaboration diagram

A.3 Component modelling

Component modelling consists of creating a *component diagram*. Such a diagram shows the dependencies between software components. A template is shown in Figure A-6.

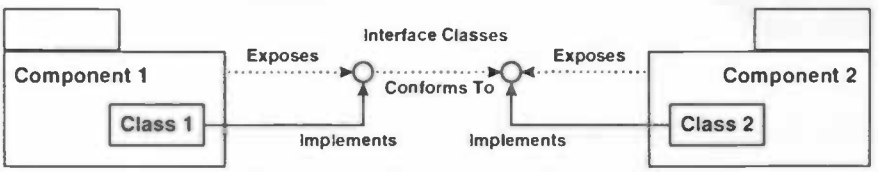


Figure A-6: Component diagram

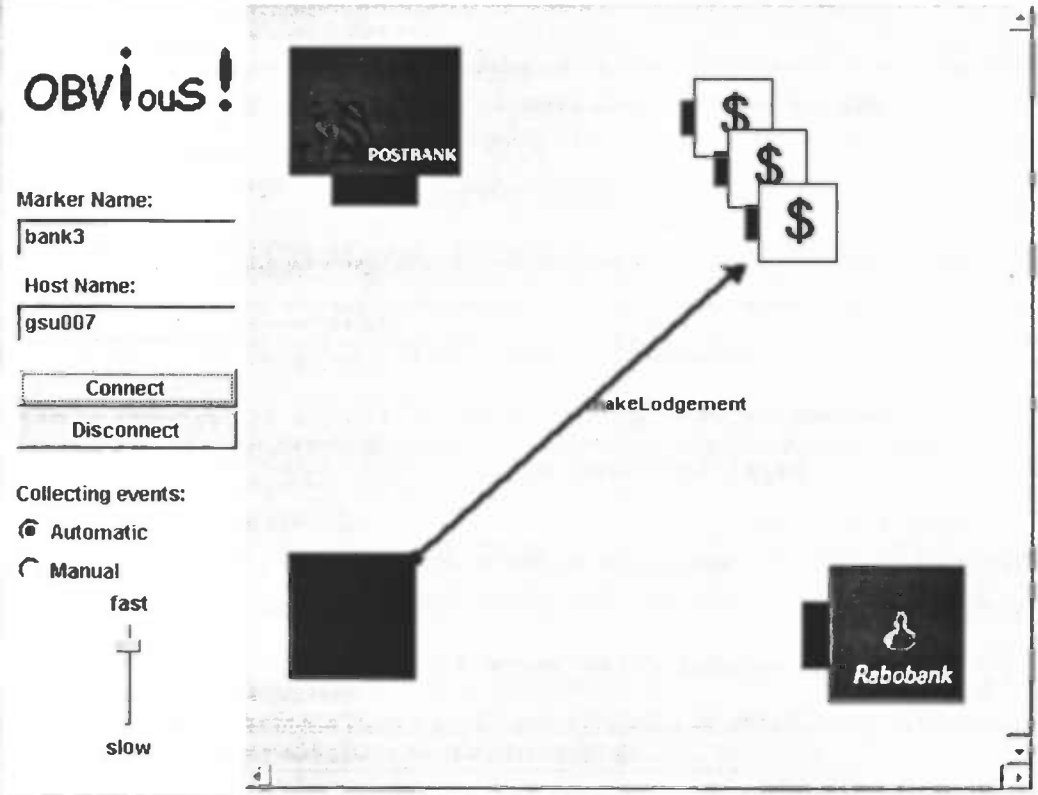
A.4 Distribution and deployment modelling

Only one diagram is drawn during this stage of modelling: the *deployment diagram*. It shows the configuration of runtime processing elements. Figure A-7 shows a deployment diagram.

Figure A-7: Deployment diagram

Applet Viewer: obvious/ObviousApplet.class

Applet



instantiate
2474
gsu007.research.kpn.com
bank
gsu007.research.kpn.com
makeLodgement
6
currentAccount

Applet started.

Appendix C An example of a Perl script

This Appendix gives an example of a simple Perl script.

```
#!perl

# this subroutine prints the argument-list
# standard routine
sub printEvent {
  # each argument has to be printed
  # it is must be followed by a newline, because of the
  # implementation of the EventServer
  if ( $currentEvent == 1 ) {
    print $nrOutputEvents, "\n";
  }
  foreach $argument (@_) {
    print $argument, "\n";
  }
  print "#endOfEvent#\n";
} # printEvent

# this subroutine parses a certain event that is passed
# through as a parameter
# it prints the event that will be generated later
# this method must be changed for own use
sub processEvent {
  # read the parameters from STDIN
  @Corba_event = <STDIN>;
  chop(@Corba_event);

  # it depends on the type of event which information
  # has to be returned
  EVENT_TYPE: {
    if ($Corba_event[0] =~ /Capsule/) {
      $nrOutputEvents = 1;
      $currentEvent = 1;
      @Vis_event = ($Corba_event[0], $Corba_event[1],
                    $Corba_event[2], $Corba_event[3],
                    $Corba_event[4], $Corba_event[5]);
      last EVENT_TYPE;
    }
    if ($Corba_event[0] =~ /Channel/) {
      $nrOutputEvents = 1;
      $currentEvent = 1;
      @Vis_event =($Corba_event[0], $Corba_event[1],
                   $Corba_event[2], $Corba_event[3],
                   $Corba_event[4], $Corba_event[5],
                   $Corba_event[6], $Corba_event[7]) ;
      last EVENT_TYPE;
    }
    if ($Corba_event[0] =~ /Interaction/) {
      # it depends on the kind of information what kind of event
      # should be returned
      OPERATION: {
        $nrOutputEvents = 1;
        $currentEvent = 1;
      if ( $Corba_event[10] =~ /IT_PING/ ) {
```

```
        @Vis_event = ("noEvent");
          last OPERATION;
      }
        if ( ($Corba_event[8] =~ /request/) &&
             ($Corba_event[1] =~ /instantiate/) ) {
          @Vis_event = ($Corba_event[0], $Corba_event[1],
                        $Corba_event[2], $Corba_event[3],
                        $Corba_event[4], $Corba_event[5],
                        $Corba_event[6], $Corba_event[7],
                        $Corba_event[8], $Corba_event[9],
                        $Corba_event[10], $Corba_event[11],
                        $Corba_event[12]);
          last OPERATION;
        }
        if ( ($Corba_event[8] =~ /reply/) &&
             ($Corba_event[1] =~ /destroy/) ) {
          @Vis_event = ($Corba_event[0], $Corba_event[1],
                        $Corba_event[2], $Corba_event[3],
                        $Corba_event[6], $Corba_event[7],
                        $Corba_event[4], $Corba_event[5],
                        $Corba_event[8], $Corba_event[9],
                        $Corba_event[10], $Corba_event[11],
                        $Corba_event[12]);
          last OPERATION;
        }
        @Vis_event = ("noEvent");
        last OPERATION;
      }
      last EVENT_TYPE;
    }
  }
  # print the result on standard output
  &printEvent(@Vis_event);
} # processEvent

$SAVEOUT = "";

sub SetOutput {
  local($fd) = shift;
  open(SAVEOUT, ">&=$fd");
  select(SAVEOUT);
}

sub CloseOutput{
  close(SAVEOUT);
}

sub CloseInput{
  close(STDIN);
  open(STDIN);
}

sub SetInput{
  local($fd) = shift;
  close(STDIN);
  open(STDIN, "<&=$fd");
}

## global variables
$nrInteractions = 0;
$nrOutputEvents;
$currentEvent;
```

Appendix D An example of a configuration file

This Appendix gives an example of a configuration file. This configuration is used with the Perl script of the previous Appendix.

```
/**
 *  1997, KPN Research and Rijksuniversiteit Groningen
 *
 *  Author: Petra Oldengarm
 *
 */

/* name of the view */
firstView

/* width and height of the template */
450 450

/* background color */
lightGray

/* capsules */
capsule
 20 40
 GridPetra
 file grid1.gif

capsule
 300 300
 Client
 file user.gif

/* interaction layout */
interaction
 rgb 37 150 119
 1

/* channel layout */
channel rgb 128 128 255
```