Αυτοματοποιημένη Ενσωμάτωση Κατανεμημένων Εφαρμογών Βασιζόμενων σε Ετερογενές Ενδιάμεσο Λογισμικό

Ιάσων Τσαπαρλής

ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

— ♦ —

Ιωάννινα, Ιούνιος 2009

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF IOANNINA

ΑΥΤΟΜΑΤΟΠΟΙΗΜΕΝΗ ΕΝΣΩΜΑΤΩΣΗ ΚΑΤΑΝΕΜΗΜΕΝΩΝ ΕΦΑΡΜΟΓΩΝ ΒΑΣΙΖΟΜΕΝΩΝ ΣΕ ΕΤΕΡΟΓΕΝΕΣ ΕΝΔΙΑΜΕΣΟ ΛΟΓΙΣΜΙΚΟ

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

υποβάλλεται στην

ορισθείσα από τη Γενική Συνέλευση Ειδικής Σύνθεσης

του Τμήματος Πληροφορικής Εξεταστική Επιτροπή

από τον

Ιάσονα Τσαπαρλή

ως μέρος των υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ

ΛΟΓΙΣΜΙΚΟ

Ιούνιος 2009

Dedication

To my parents

Acknowledgement

I would like to thank my supervisor assistant professor **Apostolos Zarras** for the opportunity he provided me to work under his supervision for carrying out the research study reported in this thesis. His constant guidance, help and encouragement during the whole process of this thesis were invaluable.

Table of Contents

List of Figures

List of Tables

List of Listings

ABSTRACT

Iason G. Tsaparlis, MSc, Computer Science Department, University of Ioannina, Greece. June 2009.
Title of Dissertation: "Automated Integration of Distributed, Based on Heterogenous Middleware Applications".
Thesis Supervisor: Apostolos Zarras

In this thesis, we propose an automated process for the *interoperability* and *integration* of distributed applications that are based on *heterogenous middleware*. In particular, for a pair of middleware platforms A and B, used by a client and a server application respectively, the proposed framework creates the illusion that the server application relies on platform A. To enable this illusion, the framework generates automatically: (i) Web services that wrap the functionality of the actual server application; and (ii) an A-specific view of the server application playing the role of the client for the Web service wrappers.

To enable the automatic code generation process, we define a *set of rules* that model the descriptions of mappings (e.g. data types, interface definitions) between the different middleware standards (e.g. CORBA and Java RMI) and WSDL. The proposed framework implements a *code generator mechanism* that accepts as input the aforementioned set of rules. This mechanism interprets the set of rules and produces the desirable source code that brings the interoperability. In this way we attain the integration, without depending on the middleware platforms assumed by the legacy applications and without interfering with their source code. In addition, the integration is transparent to both client and server applications.

We evaluate the proposed methodology both from the application developer's and final user's perspective. The benefit for a developer wishing to write the source code manually is important even in middle scale systems. On the other hand, the use of the extra software interoperable elements introduces a substantial delay in the side-to-side communication.

Εκτενης Περιληψη

Ιάσων Γ. Τσαπαρλής, MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων. Ιούνιος 2009.

Τίτλος Διατριβής: "Αυτοματοποιημένη Ενσωμάτωση Κατανεμημένων Εφαρμογών Βασιζόμενων σε Ετερογενές Ενδιάμεσο Λογισμικό".

Επιβλέπων Καθηγητής: Απόστολος Ζάρρας.

Η εξέλιξη των κατανεμημένων συστημάτων εισάγει *το πρόβλημα της ετερογένειας στο ενδιάμεσο λογισμικό* (middleware). Κατανεμημένες εφαρμογές που έχουν υλοποιηθεί με βάση διαφορετικές πλατφόρμες ενδιάμεσου λογισμικού πρέπει να διασυνδεθούν, ώστε να επιτευχθεί η *διαλειτουργικότητα*. Πρόσφατες προσεγγίσεις στο ζήτημα αυτό πετυχαίνουν μεν τον στόχο αλλά επιβάλλουν στις υπάρχουσες εφαρμογές περιορισμούς, σε σχέση με την υλοποίησή τους. Ειδικότερα, ο κώδικας των εφαρμογών πρέπει να τροποποιείται ώστε να καθίσταται συμβατός με τους μηχανισμούς που προτείνουν οι διάφορες μεθοδολογίες. Στην παρούσα εργασία ακολουθούμε μια διαφορετική φιλοσοφία αναφορικά με την επίτευξη της διαλειτουργικότητας μεταξύ ετερογενών πλατφορμών middleware. Η προτεινόμενη μεθοδολογία δεν επιβάλλει:

- κανένα περιορισμό στο είδος των προτύπων ενδιάμεσου λογισμικού που χρησιμοποιούνται για την υλοποίηση των κατανεμημένων εφαρμογών,
- καμία τροποποίηση στον πηγαίο κώδικα των υπαρχουσών εφαρμογών.

Συγκεκριμένα, για μια εφαρμογή-πελάτη που έχει υλοποιηθεί στη βάση μιας πλατφόρμας ενδιάμεσου λογισμικού A και για μια εφαρμογή-εξυπηρετητή που βασίζεται σε μια πλατφόρμα B, ο προτεινόμενος μηχανισμός δημιουργεί την "ψευδαίσθηση" ότι και η εφαρμογή-εξυπηρετητής βασίζεται στην πλατφόρμα A. Για το σκοπό αυτό, ο μηχανισμός κατασκευάζει αυτόματα: (1) υπηρεσίες διαδικτύου (Web services) που ενσωματώνουν τη λειτουργικότητα της πραγματικής εφαρμογής-εξυπηρετητή, λειτουργώντας ως πελάτες γι' αυτήν και (2) μια εικονική εφαρμογή-εξυπηρετητή, βασισμένη στην πλατφόρμα A, που καλεί η πραγματική εφαρμογή-πελάτης. Ο εικονικός αυτός εξυπηρετητής λειτουργεί ταυτόχρονα ως πελάτης για τις υπηρεσίες διαδικτύου.

Στόχος είναι ο πηγαίος κώδικας που υλοποιεί τα παραπάνω να παράγεται αυτόματα. Για το σκοπό αυτό, ορίζουμε ένα γενικό σύνολο κανόνων που περιγράφουν τον τρόπο με τον οποίο θα παραχθεί ο επιθυμητός κώδικας. Οι κανόνες αυτοί αποτελούν ένα είδος *προτύπου* (pattern) το οποίο μοντελοποιεί την αντιστοίχιση των περιγραφών (π.χ. τύποι δεδομένων, περιγραφές διεπαφών) μεταξύ υπαρχουσών πλατφορμών ενδιάμεσου λογισμικού (π.χ. CORBA, Java RMI) και WSDL. Επομένως, για το ζεύγος των εφαρμογών που ανάφεραμε παραπάνω, ο μηχανισμός απαιτεί την ύπαρξη των προτύπων A-to-WebServices και WebServices-to-B. Τα δύο αυτά πρότυπα ορίζουν γενικά το πώς θα παραχθεί ο κώδικας για τις παρακάτω οντότητες λογισμικού:

1. Την περιγραφή της διεπαφής με βάση την πλατφόρμα A που απαιτεί η εφαρμογή-πελάτης και που θα είναι όμοια με την περιγραφή της υπάρχουσας, με βάση την πλατφόρμα B, διεπαφής (πρότυπο A-to-WebServices).

2. Τα αντικείμενα που υλοποιούν την παραχθείσα, με βάση την πλατφόρμα A, διεπαφή (πρότυπο A-to-WebServices).

3. Τις υπηρεσίες διαδικτύου που καλούν τα παραχθέντα, με βάση την πλατφόρμα A, αντικείμενα και οι οποίες ενσωματώνουν τη λειτουργικότητα της εφαρμογής-εξυπηρετητή, παρέχοντας μια όμοια διεπαφή (πρότυπο WebServices-to-B).

Για τον ορισμό των προτύπων, χρησιμοποιούμε XML. Σχεδιάζουμε και υλοποιούμε έναν μηχανισμό παραγωγής κώδικα, ο οποίος δέχεται ως είσοδο το προαναφερόμενο σύνολο προτύπων, μεταφράζοντάς το στον επιθυμητό πηγαίο κώδικα που θα επιφέρει την ζητούμενη διαλειτουργικότητα. Ένας αναλυτής SAX διαβάζει τους κανόνες που περιέχονται σε ένα πρότυπο και παράγει ένα σύνολο από αντικείμενα μιας ιεραρχία κλάσεων. Αυτή η ιεραρχία κλάσεων ορίζεται σύμφωνα με το σχεδιαστικό πρότυπο Interpreter (Interpreter design pattern). Η χρήση του σχεδιαστικού αυτού προτύπου διευκολύνει τη διαδικασία παραγωγής του κώδικα. Ο κώδικας παράγεται αυτόματα και είναι έτοιμος να χρησιμοποιηθεί για την ενσωμάτωση εφαρμογών που βασίζονται σε ετερογενές ενδιάμεσο λογισμικό.

Η χρήση των προτύπων που ορίζουν τους κανόνες παραγωγής του κώδικα προσδίδει στο μηχανισμό τη δυνατότητα να επιτυγχάνει την ενσωμάτωση, ανεξαρτήτως του ενδιάμεσου λογισμικού που χρησιμοποιεί τόσο ο πελάτης όσο και ο εξυπηρετητής. Επίσης, το επιθυμητό αποτέλεσμα επιτυγχάνεται διατηρώντας την κλειστότητα του κώδικα των εμπλεκόμενων εφαρμογών, με τρόπο διάφανο προς αυτές.

Αξιολογούμε την προτεινόμενη μεθοδολογία, τόσο από την οπτική του σχεδιαστή εφαρμογών όσο και από την οπτική του τελικού χρήστη ενός κατανεμημένου συστήματος. Συγκεκριμένα, εστιάζουμε: (1) στο όφελος που αποκομίζει ένας σχεδιαστής-προγραμματιστής εφαρμογών, αφού δεν χρειάζεται να γράφει τον διαλειτουργικό πηγαίο κώδικα, και (2) στην επιβάρυνση της απόδοσης που επιφέρει αναπόφευκτα η προσθήκη των διαλειτουργικών οντοτήτων λογισμικού στο κατανεμημένο σύστημα.

Για την αξιόλογηση του οφέλους για τον προγραμματιστή, μετρούμε το πλήθος των γραμμών κώδικα *LOC* (Lines of Code) που παράγει αυτόματα ο μηχανισμός. Τυποποιούμε το LOC ως συνάρτηση μεγεθών που αντανακλούν την κλίμακα μιας εφαρμογής-εξυπηρετητή (π.χ. αριθμός των παρεχόμενων διεπαφών, αριθμός των αντικειμένων που υλοποιούν τις διεπαφές, αριθμός των παρεχόμενων λειτουργιών.). Το όφελος για τον προγραμματιστή είναι αδιαμφισβήτητο ακόμη και σε σχετικά μεσαίας κλίμακας συστήματα. Έτσι, ο προγραμματιστής κερδίζει σημαντικό χρόνο, τον οποίο μπορεί να αφιερώσει σε θέματα σχετικά με την υλοποίηση της λειτουργικότητας των εφαρμογών.

Η προσθήκη των επιπλέον στοιχείων στο κατανεμημένο περιβάλλον έχει ως αποτέλεσμα μια επιβάρυνση όσον αφορά την απόδοση των εφαρμογών που ενσωματώνονται. Η χρήση των υπηρεσιών διαδικτύου ως ενδιάμεσης πλατφόρμας για τη διαλειτουργικότητα εισάγει μια σημαντική καθυστέρηση στο χρόνο εξυπηρέτησης των εφαρμογών-πελάτη.

Chapter 1

Introduction

1.1 The Issue of Heterogeneity in Distributed Systems

A *distributed system* is a collection of autonomous computer systems that are connected through a network [6]. In each computer system, one or more applications are executed, interacting with each other and with the applications of other systems. The independence of these computer systems comes from the fact that their existence and function is independent of the existence and the function of the whole distributed system, in which they appear.

The inherent *heterogeneity* of the distributed applications can arise from differences in device architectures, data representations, communication mechanisms and programming languages. This heterogeneity imposes the use of a middleware platform that facilitates the *interoperability*, so that the distributed system appears as a single and integrated computing entity. The term *interoperability* reflects the capability of an application being executed in a computer system to obtain access to applications being executed in other systems.

Middleware is the current trend in the development of open distributed systems. It is perceived as a software layer that stands between the operating system and the applications, providing the developer with the facilities that render an application distributed [4]. Middleware consists of a basic communication mechanism, which is often called a broker, and a number of middleware services. The broker masks the differences in underlying architectures to enable the *interoperation* between the constituent elements of the distributed system.

In this way, middleware makes possible the transparent integration of distributed applications. The mechanisms offered by middleware platforms encapsulate the distributed nature of the applications. A developer does not need to care about issues regarding the distributed execution of an application. He/she is free to consider issues, related exclusively to the implementation of the functionality that the application provides. This functionality is realized by a software entity (e.g. a class) on top of a middleware infrastructure (e.g. CORBA, J2EE) and is provided through a well-defined interface.

Conventional distributed systems are build upon their typical users' requirements. All the applications within the distributed environment are implemented on top of the same middleware platform. However, the evolution of the distributed systems demands that multiple middleware systems have to be combined. Legacy or off-the-shelf applications that have been implemented on top of different middleware infrastructures have to be integrated. Suppose, for instance, that the two departments of an organization use conventional information systems that have been build on top of different middleware platforms. The workload of the organization imposes the exchange of information between these two systems. Unfortunately, this is impossible because of the different middleware infrastructures (figure 1.1). For this reason, the question that comes up is the following:

"Is middleware still the full answer to the issue of interoperability in this kind of distributed environments?"

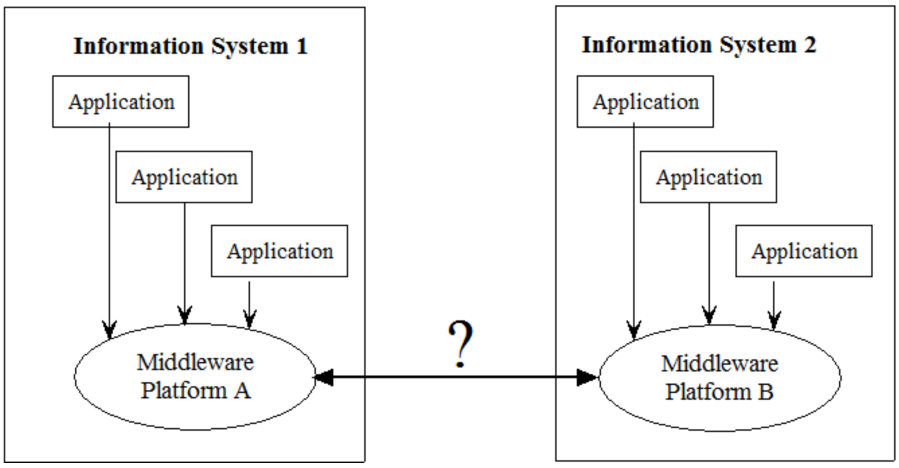


Figure 1.1: Heterogenous Middleware

In the previous case, besides heterogeneity in device architectures, communication mechanisms, data representations and programming languages, we further face *the problem of middleware platform heterogeneity*. A typical case is pervasive computing environments, which constitute a recent trend in the field of distributed systems [1]. In pervasive computing environments, the user can be anybody joining the environment with his/her mobile device. The applications deployed on the user's device can possibly assume a middleware infrastructure that differs from the one that has been employed for the development of the services offered by the environment.

Consequently, we have to make an attempt aiming at enabling the interoperation of applications, developed on top of different middleware platforms. In the recent past, some of the proposed solutions were ad-hoc, focusing on pairs of middleware platforms such as CORBA and DCOM. Other solutions, such as the middleware platforms ReMMoC [9] and JADDA [7], impose the development of client applications to rely on specific constraints, arising from these platforms. In other words, the source code of a client application must be modified. The previous is undesirable, because it contravenes the closure of the legacy application. Our purpose is to propose *a framework that enables middleware platform interoperability, without imposing any particular constraint on the middleware platforms used for the development of the distributed applications.*

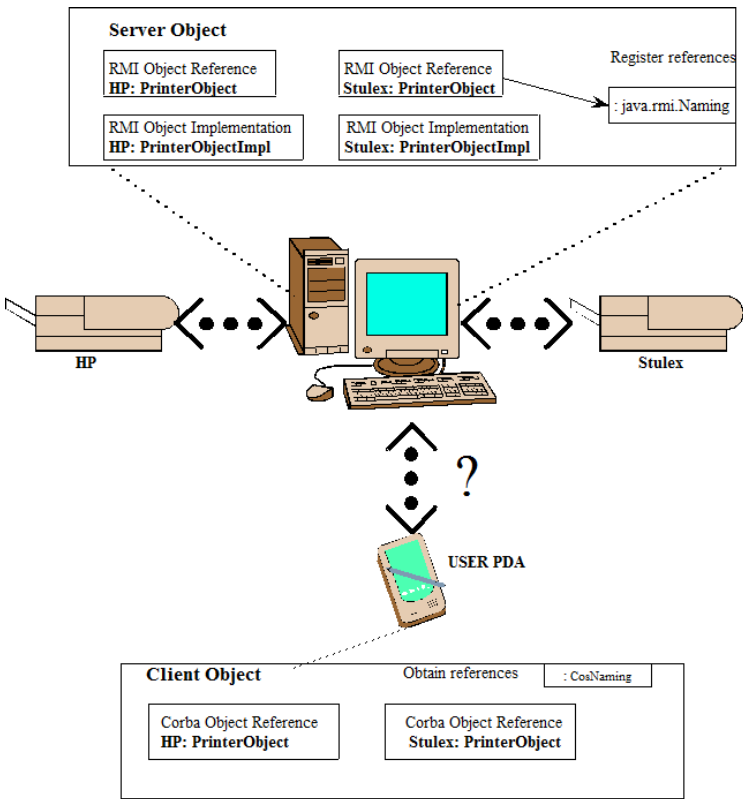

Server Object
RMI Object Reference HP: PrinterObject
RMI Object Reference Stulex: PrinterObject
Register references
RMI Object Implementation HP: PrinterObjectImpl
RMI Object Implementation Stulex: PrinterObjectImpl
: java.rmi.Naming
HP
Stulex
?
USER PDA
Client Object
Obtain references
: CosNaming
Corba Object Reference HP: PrinterObject
Corba Object Reference Stulex: PrinterObject

Figure 1.2: An Interoperability Scenario

1.2 Case Study

To highlight the concept of middleware platform heterogeneity, let us consider the following scenario (figure 1.2). Suppose that a user, carrying a PDA enters the Computer Science Department of the University of Ioannina. On top of his PDA, the user has a simple application for printing documents. The application is realized using CORBA and consists of a client object that initializes the basic CORBA broker and the CORBA Naming Service. The application uses the CORBA Naming service to locate CORBA object references to printers named "HP" and "Stulex". The expected IDL interface of these objects is supposed to provide a print() operation that accepts as input a file. The file is forwarded to the remote object, which serves as a front-end to the corresponding printer that takes in charge of printing the given file. Unfortunately for the user, the server objects that play the role of the front-end to the printers are realized on top of J2EE. The objects are pure Java RMI objects, whose references have been registered to the corresponding J2EE naming service.

1.2.1 Implementation of the Server Objects

Since the server objects are pure Java RMI objects, they realize a corresponding Java interface that provides the print() operation. Listing 1.1 gives the declaration of the interface, named *HPServer_Interf*, which provides the HP printing functionality. Each server object implements the print() method. Listing 1.2 gives the implementation of the HP server object (class *HPServer_Impl*).

Listing 1.1: A Java RMI Remote Interface

```java
import java.rmi.*;

public interface HPServer_Interf extends Remote{
  void print(String filename) throws RemoteException;
}
```

Listing 1.2: A Java RMI Implementation of a Server Object

```java
import java.rmi.*;
import java.rmi.server.*;
import java.lang.*;

public class HPServer_Impl extends UnicastRemoteObject implements
    HPServer_Interf{
  public HPServer_Impl() throws RemoteException {}

  public void print(String filename) throws RemoteException{
    // method implementation
  }
}
```

In order for the printing objects to be available to the clients, references of them must be registered to the corresponding J2EE naming service. Listing 1.3 gives the class *HPServer* that contains the main method of the HP server application. A reference of the object implementation is registered with the name "HP" to the naming service (lines 10-11). The server is now ready to accept client requests.

Listing 1.3: A Java RMI Server

```
1   import java.rmi.*;
    import java.rmi.server.*;
    import java.net.*;
    import java.lang.*;
5
    public class HPServer{

      public static void main(String args[]){
        try{
10        HPServer_Impl hpserver = new HPServer_Impl();
          Naming.rebind("HP", hpserver);
          InetAddress address = InetAddress.getLocalHost();
          System.out.println("RMI HP Server started on IP " + address.
              getHostAddress() + "\nWaiting for incoming requests...");
        }
15      catch(Exception ex){
          System.err.println(ex);
          ex.printStackTrace();

        }
20    }
    }
```

1.2.2 Implementation of the Client Object

Listing 1.4 gives the source code that implements a possible client application for our case study scenario. In the main method of the class *CorbaClient*, the Object Request Broker and the naming service are initialized (lines 16-18). Through the naming service, the application searches for a reference, using the name "HP", to a remote printing object (lines 20-22). By obtaining this reference, which is expected to be a CORBA reference, the client calls the print() method, provided by the expected CORBA remote interface (line 23). Instead, the remote interface is a Java RMI interface, as we saw in the previous subsection. Consequently, the client application will not work.

Listing 1.4: A CORBA Client

```java
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.lang.*;

public class CorbaClient{

  HPServer_Interf hp;

  public static void main(String args[]) throws WrongUsageException{
    try{

      if (args.length != 6)
        throw new WrongUsageException();

      ORB orb = ORB.init(args, null);
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("
         NameService");
      NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

      if (args[4].equals("HP")){
        hp = HPServer_InterfHelper.narrow(ncRef.resolve_str(args[4]));
        System.out.println("Obtained a handle on HP server object");
        hp.print(args[5]);
      }
      else
        throw new WrongUsageException();
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
    }
  }
}
```

1.3 Transparent Integration of the Heterogenous Applications

1.3.1 Bridging

Middleware interoperability denotes the ability of applications that have been implemented on top of different middleware infrastructures to work together. Figure 1.3 shows how the limitations discussed in the previous sections are overcome. The different middleware systems participating in application requests communicate with each other via *interoperability bridges*. A *bridge* acts as the *mediator* that masks the gap arising from the middleware heterogeneity. Generally speaking, bridges map the representation of object

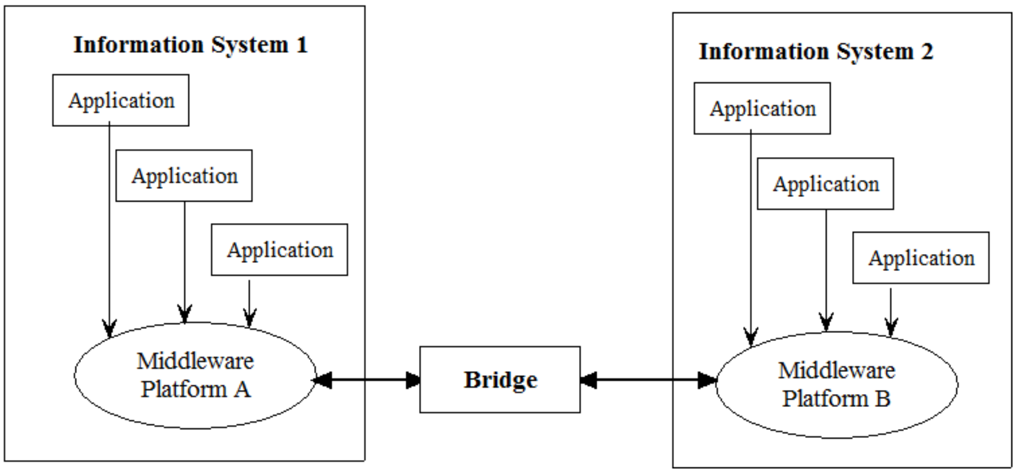


Figure 1.3: Transparent Integration of Heterogenous Middleware

references that are valid in the domain of one middleware infrastructure to references to the same object in the representation that another infrastructure assumes. In this way, the request is translated from the domain of one middleware implementation to another implementation [6]. Figure 1.3 also denotes the transparent way of integration. The fact that the server objects are connected to a different middleware, is transparent for the client objects. Likewise, it is transparent to the server objects that provide services to client objects based on a different middleware. Both the client and server objects communicate only with their middleware, as they did before, and the bridges conceal the remaining heterogeneity.

1.3.2 Implementation of the Interoperability Components

The mediator that brings the interoperability of the heterogenous middleware can be implemented by application developers on top of an existing middleware implementation. Regarding our example scenario, the mediator may consist of a surrogate object. This object is a surrogate for the server that is invoked by the client using the client's middleware. Moreover, the same object is a surrogate of the client's representation in the server's middleware. Therefore, the mediator simultaneously plays the role of a server for the actual client and the role of a client for the actual server.

Listing 1.5 gives the CORBA IDL interface that the CORBA client application of our scenario requires. The HP server object is supposed to provide this interface (*HPServer_Interf*). The necessary stub components for the implementation of a distributed application on top of CORBA arise from this language-independent interface declaration. Listing 1.6 gives the realization of the previous interface (*class HPServer_Impl*). This is the surrogate object that translates the CORBA call into a corresponding Java RMI call. We can realize easily that the *print()* method (lines 27-39) implements a Java RMI client. A Java RMI reference to a server object is looked up within the J2EE naming service (line 29).

Using this reference, the actual print() operation of the Java RMI server is invoked (line 31).

Listing 1.7 gives the implementation of the HP server application (class *HPServer*), which accepts the CORBA client requests. The CORBA server initializes the ORB and activates the POA object (lines 11-13). A reference of the CORBA-specific object implementation is registered with the name "HP" to the CORBA Naming Service (lines 14-21). The server can now accept requests.

Listing 1.5: The CORBA IDL Interface provided by the HP Server

```
interface HPServer_Interf {
        void print(in string filename);
};
```

In this way, we can achieve the transparent integration of the heterogenous applications. The CORBA client is unaware of the fact that invokes a server object which is based on Java RMI. The client is serviced as if it were a CORBA server providing the actual service. Similarly, the Java RMI server is unaware of the fact that its printing operation is utilized by a client object based on CORBA. In addition, the closure of both the client and server applications is maintained.

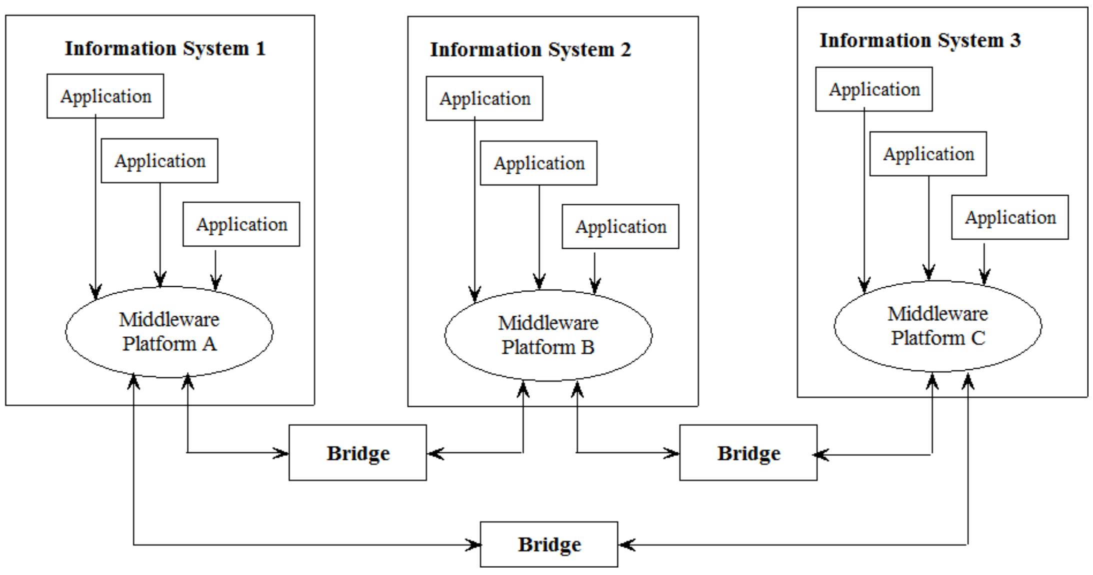


Figure 1.4: Large Scale Distributed Systems

1.3.3 Large Scale Integration and Web Services

Our scenario reflects a simple case where the extra necessary source code for a bridge can be written easily and quickly. But what about cases of large scale distributed systems with multiple middleware infrastructures being involved? Considering again the case of the organization mentioned in section 1.1, suppose that a new information system is installed whose applications have been implemented on top of a new middleware platform C. In order for the new applications to interplay with the existing ones, we have to implement further mediators. In a similar manner to the one described in the previous subsection, a developer must implement mediators that bridge the gap between middleware platforms A and C, and between platforms B and C. Figure 1.4 shows this interoperability scenario.

As the scale of the distributed system increases and as new middleware platforms are released in the market, we realize that bridging the heterogeneity in the previous way becomes a very complicated procedure. We always have to map the descriptions of each existing middleware implementation to those of the new one. It would be desirable to find a common reference base on which all the mappings should rely. Both the existing and the possible new middleware infrastructures will be mapped according to this reference base.

Web services[1] can provide us such a reference base. They were originally proposed to wrap conventional business information systems (BISs), developed on top of different middleware platforms. They are stateless software entities whose interface is specified using WSDL, a commonly agreed XML-based language. Accessing Web services relies on SOAP, a commonly agreed XML-based message format, and other widely accepted standards such as HTTP and SMTP.

Instead of accessing directly the actual server object (see Listing 1.6), the surrogate object can play the role of a Web service client that invokes a Web service. This Web service acts as client for the actual server object, wrapping its functionality. The Web service wrapper is the front-end for the server object. Figure 1.5 shows the interoperability scenario for our casy study, in case of using Web services for the integration.

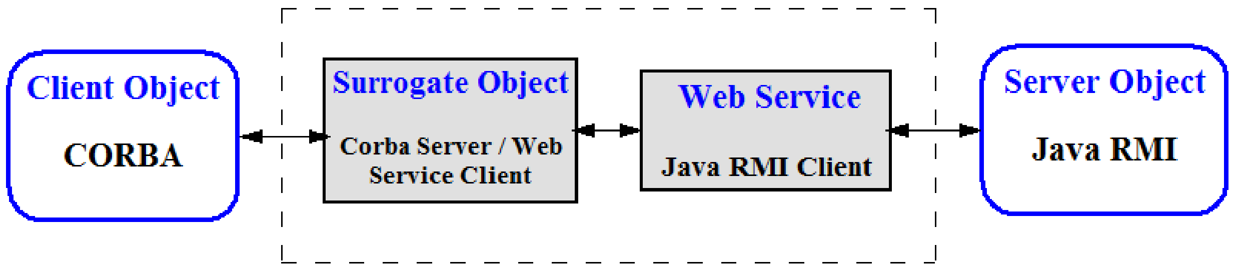


Figure 1.5: Integration using Web Services

[1]http://www.w3.org/TR/ws-arch/

Listing 1.6: The CORBA-specific Object Implementation acting as a Java RMI Client

```java
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.rmi.*;
import java.rmi.server.*;
import java.util.Properties;
import java.net.*;
import java.lang.*;


public class HPServer_Impl extends HPServer_InterfPOA {
        private ORB orb;
        private String RMIServerURL;
        HPServer_Interf ref;

        public HPServer_Impl(String servURL) {
                RMIServerURL = servURL;
        }

        public void setORB(ORB orb_val) {
                this.orb = orb_val;
        }


        public void print(String filename) {
                try {
                        ref = (HPServer_Interf) Naming.lookup(RMIServerURL)
                            ;
                        System.out.println("Obtain a handle on RMI HP
                            server object");
                        ref.print(filename);
                }
                catch(Exception ex) {
                        ex.printStackTrace();
                        System.out.println("Cannot establish connection
                            with RMI server!");
                        System.exit(0);
                }
        }

}
```

Listing 1.7: The CORBA HP Server that the Client invokes

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
import java.net.*;

public class HPServer {
        public static void main(String args[]) {
                try {
                        ORB orb = ORB.init(args, null);
                        POA rootpoa = POAHelper.narrow(orb.
                            resolve_initial_references("RootPOA"));
                        rootpoa.the_POAManager().activate();
                        org.omg.CORBA.Object objRef = orb.
                            resolve_initial_references("NameService");
                        NamingContextExt ncRef = NamingContextExtHelper.
                            narrow(objRef);
                        HPServer_Impl impl = new HPServer_Impl("rmi
                            ://192.168.0.2/HP");
                        impl.setORB(orb);
                        org.omg.CORBA.Object ref = rootpoa.
                            servant_to_reference(impl);
                        HPServer_Interf href = HPServer_InterfHelper.narrow
                            (ref);
                        NameComponent path[] = ncRef.to_name("HP");
                        ncRef.rebind(path, href);
                        System.out.println("Corba HP started on IP " +
                            InetAddress.getLocalHost().getHostAddress()+"\n"
                            );
                        orb.run();
                }
                catch(Exception ex) {
                        System.err.println("ERROR: " + ex);
                        ex.printStackTrace(System.out);
                }
        }
}
```

In this way, the surrogate object translates the CORBA client's request into a Web Service call, which is then translated into a Java RMI request. If we consider the distributed system of the organization (figure 1.3), for a client application on top of platform A and a server application on top of platform B, we need to map the description of infrastructure A to a Web service description (A-to-WebServices mapping) and the Web service description to a description of infrastructure B (WebServices-to-B mapping). For the opposite case, we need the B-to-WebServices and WebServices-to-A mappings. As the applications on top of the new platform C become part of the whole system (figure 1.4) we will need only the C-to-WebServices and WebServices-to-C mappings in order for them to interwork with the existing applications, acting either as clients or servers. On the contrary, if we do not use Web services, as a reference base middleware, we will need the A-to-C, C-to-A, B-to-C and C-to-B extra mappings.

Listing 1.8 gives the CORBA-specific object implementation of the interface required by the client application of our scenario, which acts as a Web service client. The print() method constructs a dynamic call to the Web service that acts as a front-end for the HP server object. Listing 1.9 gives this Web service wrapper, whose print() method implements a Java RMI client, calling the print() method in the actual HP object.

Listing 1.8: The CORBA-specific Object Implementation acting as a Web Service Client

```
 ...

import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;
import javax.xml.rpc.ParameterMode;
import java.util.Properties;
import java.net.*;
import java.lang.*;

public class HPServer_Impl extends HPServer_InterfPOA {
        String endpoint;
        ...

        public void print(String filename) {
                try {
                        Service service = new Service();
                        Call call = (Call) service.createCall();
                        call.setTargetEndpointAddress(new URL(endpoint));
                        call.setOperationName("print");
                        call.addParameter("arg1", XMLType.XSD_STRING,
                           ParameterMode.IN);
                        call.setReturnType(XMLType.AXIS_VOID);
                        call.invoke(new java.lang.Object[] { filename } );
                }
                catch(Exception ex) {
                        System.err.println("ERROR: " + ex);
                        ex.printStackTrace(System.out);
                        System.out.println("Cannot establish connection
                           with Web Service!");
                }
        }

}
```

Listing 1.9: The Web Service Wrapper acting as a Java RMI Client

```
import java.rmi.*;
import java.net.*;

public class WS_HP_RMIClient {

        HPServer_Interf objref;

        public void print(String filename) {
                try {
                        InetAddress address = InetAddress.getLocalHost();
                        String serverURL = "rmi://" + address.
                           getHostAddress() + "/HP";
                        objref = (HPServer_Interf) Naming.lookup(serverURL)
                           ;
                        System.out.println("Obtain a handle on RMI HP
                           server object");
                        objref.print(filename);
                }
                catch(Exception ex) {
                        ex.printStackTrace();
                        System.out.println("Cannot establish connection
                           with RMI Server!");
                        System.exit(0);
                }
        }
}
```

1.4 Illusion Maker: Automating the Integration of Heterogenous Middleware

We have already emphasized that by using Web services, we can facilitate the interoperability between different platforms, in a scalable manner. However, in a case of a large system it will be quite demanding for the application developers to write manually the source code that leads to the interoperability. Not only have they to specify the mappings of the descriptions between the heterogenous implementations each time, but also they have always to write customary code, irrelevant to the functionality of the applications.

A promising idea is to try to model the mappings between the different middleware standards according to a *set of rules*. This set of rules must define the mappings in an abstracted and coherent form. Then, by applying these rules, the purpose is to automate the procedure of generating the necessary source code for the integration. In this way, the developer will obtain automatically the interoperable source code, being free to devote more attention to issues regarding the implementation of the functionality.

In this thesis, we propose *a framework that enables middleware platform interoperability, without imposing:*

- *any requirements on the middleware platforms used for the development of either the client or server applications;*
- *any code modifications.*

Specifically, *for every client application that relies on a middleware platform X, the proposed framework creates the illusion that the server applications rely on the same platform.* To enable this illusion, the framework automatically generates (1) Web services that wrap the functionality of the server objects, and (2) a X-specific view of the server object, playing the role of the client for the Web service wrappers. For this reason, we call the proposed framework *"The Illusion Maker Framework"*. Considering our example scenario, the Illusion Maker will automatically generate the source code, presented in Listings 1.5, 1.7, 1.8, 1.9, creating the illusion that the Java RMI server application relies on the CORBA platform, on which the client application actually relies.

The automatic code generation process is based on a set of rules that model the descriptions of mappings between the different middleware standards, mentioned in the previous subsection. This involves mapping the build-in types and the interface definitions of each middleware platform (such as CORBA and J2EE) to WSDL, and vice versa. We call this set of rules *"The platform specific patterns"*, because they specify how to create the aforementioned platform specific "illusion". For a pair of platforms A and B assumed by a client and a server respectively, the Illusion Maker requires the existence of 2 corresponding patterns, A-to-WebServices and WebServices-to-B. The patterns specify generically how to generate:

1. The A-specific illusion interface, required by the client application (pattern A-to-WebServices).
2. The A-specific implementations of the objects that serve as Web service clients (pattern A-to-WebServices).
3. The Web services that serve as B-specific clients to the actual server objects (pattern WebServices-to-B).

The proposed framework implements a *Code Generator Mechanism* that accepts as input the platform specific pattern. The mechanism interprets the rules within the provided pattern and produces the corresponding source code. This automatically generated source code is ready to be utilized for the integration of the applications based on heterogenous middleware. We can define this kind of rules for every possible pair of middleware platforms assumed by a client and a server application respectively. By using the Illusion Maker framework, we attain the integration without depending on the assuming middleware platforms and without touching the source code of the legacy applications.

1.5 The Structure of the Thesis

This thesis is structured as follows:

- Chapter 2 discusses the background about the middleware that facilitates the design and development of distributed systems. It presents further related work with respect to the issue of middleware platform interoperability.

- Chapter 3 details the basic concepts of the proposed framework. It describes the input components of the Illusion Maker framework, and elaborates the definition of the platform specific patterns. A detailed view of the design and the implemantion of the Code Generator mechanism is also given.

- Chapter 4 presents an evaluation of our proposed methodology, both from the developer's and from the final user's perspective. An application developer reaps the benefit arising from the automated code generation process. On the other hand, a user of a distributed application may realize a decline on the performance due to the communication overhead, which is introduced inevitably.

- Chapter 5 summarizes the contribution of the current thesis and discusses possible future work.

Chapter 2

Background and Related Work

This chapter starts with the fundamentals of middleware that facilitate the design and the development of distributed systems. Afterwards, it presents a review of the related work, in the direction of middleware platform interoperabilty. Finally, we point out again the purpose of our proposed methodology and how its contribution is diversified from the previous work.

2.1 Middleware and Distributed Systems

Middleware [4] is a software layer that stands between network operating systems and distributed applications. It is in charge of issues concerning the execution of an application within a distributed environment. It releases the developer from this obligation and permits him/her to take action only about issues regarding the implementation of the functionality provided by distributed applications. Middleware realizes the encapsulation of the distributed nature of applications. This feature of distributed systems is called *distribution transparency* [6] [16]. Thus, the developer implements the functionality of the distributed application, as if it were for an application intended for a non-distributed environment.

The communication between the distributed applications is carried out through a computer network. The network is responsible for the physical transmission of data as electrical signals. The data is routed to the destinations as a set of packets, composed of byte streams or fixed-length messages. This network functionality is provided by the operating system. If the distributed applications are build upon directly the operating system, the developer will confront issues related to the communication between the applications. These issues arise from the heterogenous nature of the involved computing entities (e.g. differences in data representations, incompatibilities in data types, synchronization between client and server, error detection, etc.). Bridging this gap by the application engineer is time consuming, error prone, and distracts his/her attention from the problems related to the provided functionality.

Middleware comes to overcome this conceptual gap. It provides the application engineer with a high level of abstraction, based on primitives that are provided by the network operating system. In this way, it hides the complexity of using a network operating system from application developers. Without this, the bridging of this gap would be difficult to achieve.

2.2 Basic Properties of Middleware

The utilization of middleware for the design and development of distributed systems gives them some properties concerning their quality and their functionality. Because these properties are fundamentally substantial for the distributed systems, they form an important part of the International Standard on Open Distributed Processing (ODP) [11]. As far as quality is concerned, a middleware platform should provide a distributed system with *openess*, *scalability*, *performance* and *fault-tolerance*. Regarding functionality, a middleware platform should provide a distributed system with *distribution transparency* [6] [16]. In the following subsections, we discuss these basic properties.

2.2.1 Openness

Openess means that the system can easily be modified in the presence of changing functional requirements, with the existing architecture remaining stable. To achieve this, the software components of the system must have well-defined interfaces. A client component obtains access to a server component through its interface. The implementations of the interfaces are completely encapsulated from the components. Changes in the functional requirements must be implemented as extensions of the existing interfaces or must be derived from new interfaces. The existing implementations must never be touched. If the previous happens, all the components depending on the modified implementations will be affected. The existing middleware infrastructures enable us to declare interface definitions, through specific descriptions called *Interface Definition Languages* (IDLs).

2.2.2 Scalability

Scalability reflects the ability of a system to handle successfully future growing loads. For instance, if the number of requests demanding service grows rapidly, the system should be able to accommodate the growing demands by maintaining simultaneously its behavior stable. To achieve this, extra software components can take in charge of servicing the increasing number of requests. The architectural design which is based on the utilization of the interfaces facilitates such a solution. In addition, *trading services* can contribute to the scalability of a system, as a load balancing mechanism.

2.2.3 Performance and Fault-Tolerance

Performance denotes the efficient execution of the applications that are built on top of a middleware infrastructure. *Fault-Tolerance* demands that a system should continue to operate, even in the occurrence of faults. It would be desirable to achieve fault-tolerance with the limited interference of users or system administrators.

2.3 Distribution Transparency

Distribution transparency involves the encapsulation of the fact that a system is composed from distributed components. This is the reason why a distributed environment is perceived by users and developers as an integrated entity, rather than as a collection of independent elements [6] [16]. Distribution transparency is refined to a number of more specific transparencies, which are described in the following subsections.

2.3.1 Access Transparency

From the developer's perspective, access transparency demands that accessing either local or remote application components should be obtained in the same way. In other words, the interface provided by a server component is the same for the communication between components on the same host and components on different hosts. Middleware platforms provide access transparency through the utilization of *stubs*. A stub acts as a surrogate or a proxy of the server component to the client component, and vice versa. The stubs hide the fact the a service request is remote. The client component makes a local call to the stub-proxy of the server component, which in turn forwards the request to the server component, via the network. In a similar manner, the server-side proxy of the client component accepts the client's request, and converts it into a local service request. An identical procedure takes place for the transfer of the server's response back to the client component.

The two stubs translate the request and the response into a suitable form, for their transmission through the network and then again to a perceivable form for the components. In this way, the gap which is due to the heterogeneous features of the system components is bridged. Middleware platforms employ special compilers for the automatic stub generation coming from the corresponding IDL descriptions. The stubs provide interfaces which can be utilized in the source code of a client component, in order that access transparency be achieved.

2.3.2 Location Transparency

Location transparency enables a client component to discover and access server components, without knowledge of their physical location. Middleware infrastructures provide this mechanism through *naming services* and *trading services.*

Naming services maintain a registry of service names, which are associated with a reference to the corresponding componenent that offers the service. Server components register this kind of associations, through interfaces provided by the naming service. Client components, through these interfaces, use the registered names to discover references to the server components that provide a desirable service. By acquiring such a reference, the client component can request a service.

Trading services provide interfaces through which a server component can register the provided operations and their quality features (i.e. response time, availability of resources, etc). A client component uses the trading service to discover server components that provide services having particular quality features. If an appropriate component is found within the registry, the client obtains a reference to this component, through which the client requests the service.

2.3.3 Concurrency Transparency

Concurrency transparency means that a server component shares its functionality with multiple client components concurrently, and the integrity of the shared component is preserved. Neither users nor developers realize how the concurrency is achieved.

Concurrency control is typically provided by the middleware platforms through *locking mechanisms* and *threads.* Locking mechanisms ensure that only one among all the client components requesting the same service can obtain access to the server component that offers the service. In case of threads, a single thread can carry out the service provision in a serial manner or multiple threads can provide the service in parallel. In the second case, the synchronization of threads that carry out the service provision, is required.

2.3.4 Failure Transparency

A distributed system should be tolerant to the presence of failures. It should continue to function properly, so that both the users and the developers be unaware of how the failures are resolved. Fault tolerance is attained through mechanisms that maintain replicas of the system components. If for some reason (e.g. software or hardware failure) a component fails to provide service, a replica of it can undertake the service provision. Middleware platforms provide interfaces for the failure detection and recovery. The mechanism which is responsible for the handling of replicas organizes each primary component with its replicas into groups. In an occurrence of a failure, a notification is sent to the recovery mechanism that takes upon to activate a replica of the failing component, so as to execute the request.

2.3.5 Migration Transparency

Sometimes it is necessary to move a component from one location to another. This may happen because of an overload of the initial host, or in order for an upgrade to take place; in addition, the component may be relocated closer to its usual clients. This kind of relocation is referred to as *migration*, and must be provided in a transparent way. This means that the service provision is executed without the client component knowing that it is being served by a component whose location has changed. The mechanisms that provide migration transparency must be able to handle the states of the components and their dependencies on other residing components. The interfaces of these mechanisms must offer operations for getting the state of a component and setting it again, after its migration. If dependencies on other components are discovered, the depending components must also migrate to the new location.

2.3.6 Persistence Transparency and Transaction Transparency

Persistence transparency refers to the mechanisms that a middleware infrastructure should provide for the storage of the components' state. The provided interfaces enable the creation of storage entities and their association with the system components. *Transaction transparency* provides means for coordinating the execution of atomic transactions.

2.4 Interaction between Distributed Components

RM-ODP (Reference Model for Open Distributed Processing) [11] proposes a generic architectural style that should be followed by middleware platforms, to enable the development of open distributed systems. According to this style, a distributed system consists of *basic engineering elements*, organized into *capsules* for the purpose of encapsulation of processing, storage, and request flow. A basic engineering element provides one or more interfaces, consisting of operations that can be invoked by other basic engineering elements. The interoperation between engineering elements that belong to different capsules is realized through *channels*. A channel is a compound element consisting of pairs of stubs, binders, and protocol elements, that realize the access transparency mechanism. As we have seen already, stubs mask the differences with respect to the heterogenous features. Binders are in charge of preserving the integrity of the channel (i.e. error detection, data multiplexing, etc.) while protocols provide the basic communication mechanisms (i.e. TCP/IP socket creation, broadcast of requests after a time-out, etc.).

Invoking an operation on a target element involves holding a reference to that element. If both the requester and the requested element reside in the same address space (i.e. elements belonging to the same capsule), the reference is a typical language-specific pointer (e.g. a C++ pointer or a Java reference). On the other hand, if the requester and the requested element reside in different address spaces (i.e. elements belonging to different capsules), the reference is a pointer to a representative (i.e. a stub-proxy) of the requested element in the requester's address space.

The typical behavior of a capsule that plays the role of a *server* is summarized in the following steps:

1. The server capsule initializes the core broker;
2. A reference to a standard middleware service that provides location transparency is obtained.
3. The basic engineering elements that constitute the server capsule are created.
4. References to the basic engineering elements are created.
5. The references are registered to the middleware service which provides location transparency, using a particular key, such as a *name*, or the *interface specification* of the referenced elements.
6. Hereafter, the references can be discovered by client capsules that wish to use the referenced elements.

The typical behavior of a capsule that plays the role of the *client* to basic engineering elements, deployed on another capsule that plays the role of the *server*, is summarized in the following steps:

1. The client capsule initializes the core broker and obtains a reference to a standard middleware service that provides location transparency.
2. The aforementioned service is used to obtain references to the basic engineering elements that should be invoked.
3. The references found in the previous step are used for invoking operations on the target elements.

Several standard middleware infrastructures such as CORBA[1], J2EE[2] and DCOM[3] follow the generic RM-ODP style [16]. In this way, a distributed system, which is built upon one of these infrastructures, integrates heterogenous applications and achieves the interoperability that renders it an integrated entity.

2.5 Related Work

In this section, we return to the issue of *middleware platform interoperability*. We refer to three approaches that deal with the issue under consideration, and we point out concisely how they achieve the interoperability.

2.5.1 JADDA: Java Adaptive framework for Dynamic Distributed Architecture

The *JADDA* [7] framework combines ideas from the fields of Dynamic Software Architectures and of Aspect-Oriented Programming to specify and implement middleware variability at both development and run-time. All the constituent architectural elements of a distributed system (e.g. software components, middleware platforms) are treated as variants that can easily be changed during development or run-time.

The framework relies on architectural specifications defined with xADL (XML-based Architecture Description Language). It constitutes XML descriptions that provide a means of standardization for expressing architectural specifications. At the implementation level, the JADDA class provides an API, which is used by the system components for service requests. The DistributedConnector abstract class is the common interface for different middleware platforms. A number of subclasses of the previous class implement the functionality of the various middleware (e.g. CorbaConnector, SoapConnector). Each specific subclass provides a call() method that transforms invocations into messages conforming with the middleware platform used for the realization of the services. The invocation is constructed at run-time, using Java Reflection.

The interoperability is attained as follows: A client wishing to request a service initializes a JADDA instance and invokes the call() method. Depending on the values of the parameters provided to the method, a search takes place among the available xADL-based architectural descriptions. This search discovers that the requested service is realized, for instance, on top of CORBA. Therefore, a CorbaConnector object is initialized at run-time, providing a reference to the CORBA Naming Service. After this point, the remote call is carried out according to the CORBA communication mechanisms. If the search discovers

[1]http://www.omg.org/technology/documents/formal/corba_iiop.htm
[2]http://java.sun.com/j2ee/
[3]http://www.microsoft.com/com/default.mspx

that the requested service has been implemented using Web services, a SoapConnector instance is instantiated dynamically providing a reference to the UDDI registry.

We perceive that middleware interoperability is transparent to the server components, but the client component must be compatible with the JADDA API. It must invoke the call() method of the JADDA object. In this way, each client is transformed dynamically into a client for the middleware platform, on top of which the server component is based.

2.5.2 ReMMoC: Reflective Middleware for Mobile Computing

ReMMoC [9] is a similar framework to JADDA. It provides a generic API, called Binding Abstraction. This API is based on WSDL, and is utilized for the discovery and request of services provided by a distributed system. The underlying Binding Framework consists of a number of different personalities. A personality is a plug-in that transforms the generic invocation made through the Binding Abstraction into a message, whose format conforms with the middleware platform that was used for the realization of the invoked service. The suitable personality is selected dynamically at run-time, through the reflection mechanism.

In addition to the Binding Framework, the Binding Abstraction provides also the Service Discovery Framework. Its personalities implement different service discovery protocols (SDPs). The mechanism looks up for services that can be found independently of the assumed SDP. For every SDP that can be discovered (based on the SDP-specific personalities currently plugged in), the framework makes either a synchronous request or monitors continuously the environment and generates an event on detection. The information related to the requested service is retrieved by the Service Discovery Framework, and then is utilized by the Binding Framework to make the remote call.

Currently, ReMMoc provides personalities for Java RMI, CORBA and SOAP. Concerning the SDPs, it incorporates personalities for SLP and UPnP. Again, the middleware platform interoperability is transparent for the actual server applications. However, a client application must utilize the ReMMoC API in order to communicate with a server. Similarly with JADDA, the client is automatically configured with respect to the middleware infrastructure assumed by the server.

2.5.3 INDISS: Interoperable Discovery System for Networked Service

INDISS [5] is a system which provides SDP interoperability in a transparent way, both for clients and servers. The interoperability mechanism is based on common features that all the SDPs share. Its architecture consists of two basic subsystems, the parser and the composer. The coordination between the two subsystems is based on an event-driven mechanism. The parser accepts the client's request and processes it, to retrieve the SDPs-independent semantics that are necessary for making feasible the interoperability. These semantics are exported from the parser as a series of events, which then feed the composer. This subsystem transforms the series of events into an integrated message, based on the server's SDP semantics. In this way, the initial data which was suitable for the client's SDP is converted into a valid form for the server's SDP.

The interaction between each couple of parser and composer is based on the event-driven architecture, which is completely hidden from the external environment of INDISS. Both the client and the server do not realize the fact that they interact, even if they are based on different SDPs.

2.6 The Contribution of the Illusion Maker Framework

In the previous section, we alluded to three frameworks that try to overcome the middleware platform heterogeneity, enabling the transparent interoperability with respect to clients and servers. While they achieve this interoperability in a transparent way, they impose the client applications to comply to the frameworks specifications. In other words, the implementations of the client applications must be modified according to the API that each framework provides. In the case of JADDA, the client application has to utilize the JADDA object, while in the case of ReMMoC the provided API must be used for the dynamic configuration of the Binding Framework. Concerning INDISS, although the SDP interoperability is transparent to both sides, the source code must be modified according to INDISS requirements for accessing the remote services.

There exist cases where the modification of the client's source code is impossible. Users may not possess the source code of their applications, so as to make this code compatible with mechanisms, such as the ones described above. Even if the source code is at the user's disposal, it will be difficult and time-consuming for a developer to modify it. For this reason, the *Illusion Maker framework* follows a different philosophy, by not imposing any constraints on the development of client and server applications. Instead, we automatically generate everything that is required to render the software parts interoperable without interfering with their source code.

Chapter 3

The Architecture of the Illusion Maker Framework

3.1 Basic Functionality

Figure 3.1 gives an overview of the general functionality of the Illusion Maker Framework. The Illusion Maker can be considered as a black box whose main functionality is to generate automatically the necessary source code for integrating applications which are based on heterogenous middleware platforms. The input needed to be given for this procedure consists of two components, the platform independent information and the platform specific pattern.

The platform independent information describes the architecture of a legacy server application, in an abstracted form (e.g. the names of the remote objects, their interface definitions, etc.). This information is necessary in order to create illusion server capsules, with respect to the client's application middleware platform, that will wrap the functionality of the actual remote objects.

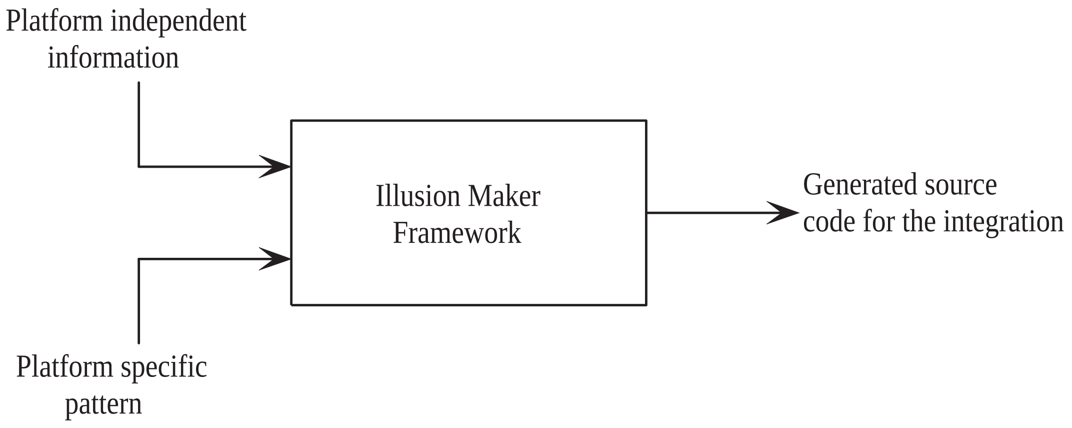

Platform independent information
Illusion Maker Framework
Generated source code for the integration
Platform specific pattern

Figure 3.1: The Basic Functionality of the Framework.

The platform specific pattern specifies the generic rules for generating the desirable platform interoperable source code. Being combined with the platform independent information, these rules are refined into the source code, through which the integration will be attained.

The following sections discuss explicitly the structure of the input and the design of the architecture of the Illusion Maker framework.

3.2 Input and the Use of XML

To specify formally the structure of both of the input components, we chose XML. *XML* (*eXtensible Markup Language*) is a general markup language which provides a basic structure and set of rules for modelling and structuring data in a completely consistent and customizable manner. Not only is it simple but also efficient and highly structured. This is why it is perfectly suited for data exchange between different application infrastructures, providing a high level of interoperability [10] [12].

In the Illusion Maker, XML has facilitated the following actions:

1. the specification of a structure which describes the platform independent information;
2. the specification of a structure for the platform specific patterns, which provide the rules for the code generation process;
3. the implementation of the whole functionality provided by the framework.

3.3 Description of the Architecture of a Legacy Server

The first input component of the framework contains information which is related to the architecture of a legacy server application. This information is independent of the middleware infrastructure assumed by the server application, and will be used for the creation of illusion server capsules, as a front-end for the actual remote application. So, it is necessary for these data to be registered in a structured manner in order to be easily accessible and retrievable. As we mentioned in section 3.2, a XML document is a suitable component for storing information in a fully structured way.

Before we describe explicitly the structure of this particular XML document, let us discuss briefly what exactly piece of information should be contained in the XML document. For this purpose, we have to consider the basic functionalities provided by the middleware platforms for the development of distributed systems and the structure that these systems are assumed to have.

In section 2.4 we have already alluded to the generic architectural style which RM-ODP has proposed towards the standardization of the basic functionalities that should be provided by middleware platforms [11]. Our approach is based on the RM-ODP proposed style, but assumes a more simple structure.

The Diagram in figure 3.2 gives an overview of our approach, concerning the architecture of a server application. The basic component is the *element* which reflects the functionality that the application offers to the distributed environment. In other words, the element is the actual remote object, which encapsulates the implementation of its functionality. Each element provides one or more *interfaces*, consisting of *operations* which can be invoked by other elements. The interface is an abstract description of the functionality of an application that provides a means of communication between distributed objects. A client application through an interface can obtain access to the server application and invoke an operation.

The elements are organized into *capsules* for the purpose of encapsulation of processing, storage and request flow. When the elements that constitute the server capsule are created, the corresponding *references* are registered to the middleware service that provides location transparency, using a particular key such as a *name*. Each element must maintain such a reference in order that other elements from client capsules be able to discover the former and use their functionality.

According to the simplified generic architectural style described above, we create specific XML documents containing the necessary information to be used as input to our framework. Listing 3.1 declares the XML schema, on which such XML documents should rely. As we can see, the root element, named *architecture* (lines 3-12), consists of a sequence of three child elements:

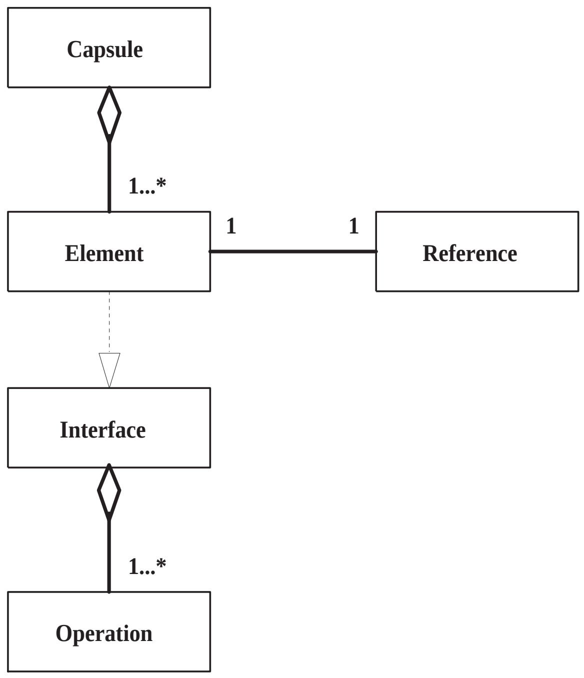


Figure 3.2: The Generic Architecture of a Server

- *interfaces*
- *elements*
- *capsules*

It also has an attribute named *type*, which according to its specific declaration (lines 32-36) its value must always be equal to "*architecture*". We can realize easily that each one of the aforementioned child elements is referred to each one of the basic components of the architectural style (figure 3.2).

The *Interfaces* element (lines 14-18) contains a number of child elements named *interface*, each one of them represents an interface that a remote object provides. The *Elements* element (lines 20-24) contains child elements named *element*, and each one represents an element-object that realizes a corresponding interface. The *Capsules* element (lines 26-30) includes child elements named *capsule*, depicting the corresponding server capsules.

For each interface, element and capsule, we have to keep some essential information which is independent of the middleware infrastructure. To achieve this we make use of the XML capabilities, that is, child elements and attributes. Therefore, for each interface (lines 38-45) we have to know its name (*name* child element) and the provided operations (*operations* child element). It may also be vital to keep in mind the assuming middleware platform. So, we use the *mdw_platform* element and currently we assume three possible values: *Corba*, *RMI* and *WS*, representing the corresponding middleware platforms (lines 64-70).

Listing 3.1: XML Schema for the Platform Independent Information

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="architecture">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="interfaces" type="interfsType" minOccurs="1"
            maxOccurs="1"/>
        <xsd:element name="elements" type="elemsType" minOccurs="1"
            maxOccurs="1"/>
        <xsd:element name="capsules" type="capsType" minOccurs="1"
            maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="type" type="archType"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:complexType name="interfsType">
    <xsd:sequence>
      <xsd:element name="interface" type="interfType" minOccurs="1"
          maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="elemsType">
    <xsd:sequence>
      <xsd:element name="element" type="elemType" minOccurs="1" maxOccurs="
          unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="capsType">
    <xsd:sequence>
      <xsd:element name="capsule" type="capsuleType" minOccurs="1"
          maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:simpleType name="archType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="architecture"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="interfType">
    <xsd:sequence>
      <xsd:element name="mdw_platform" type="mdwtype" minOccurs="1"
          maxOccurs="1"/>
      <xsd:element name="name" type="xsd:string" minOccurs="1" maxOccurs="1
          "/>
      <xsd:element name="operations" type="opsType" minOccurs="1" maxOccurs
          ="1"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:integer" use="required"/>
  </xsd:complexType>

  <xsd:complexType name="elemType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="id" type="xsd:integer" use="required"/>
        <xsd:attribute name="interf_id" type="xsd:integer" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <xsd:complexType name="capsuleType">
    <xsd:sequence>
      <xsd:element name="references" type="refsType" minOccurs="1"
          maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:integer" use="required"/>
    <xsd:attribute name="IPaddress" type="xsd:string" use="required"/>
  </xsd:complexType>

  <xsd:simpleType name="mdwtype">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Corba"/>
      <xsd:enumeration value="RMI"/>
      <xsd:enumeration value="WS"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name="opsType">
    <xsd:sequence>
      <xsd:element name="operation" type="opType" minOccurs="1" maxOccurs="
          unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="refsType">
    <xsd:sequence>
      <xsd:element name="reference" type="refType" minOccurs="1" maxOccurs=
          "unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="opType">
    <xsd:sequence>
      <xsd:element name="args" type="argsType" minOccurs="1" maxOccurs="1"/
          >
        </xsd:sequence>
88      <xsd:attribute name="id" type="xsd:integer" use="required"/>
        <xsd:attribute name="retType" type="xsd:string" use="required"/>
90      <xsd:attribute name="name" type="xsd:string" use="required"/>
      </xsd:complexType>

      <xsd:complexType name="refType">
        <xsd:simpleContent>
95        <xsd:extension base="xsd:string">
            <xsd:attribute name="id" type="xsd:integer" use="required"/>
            <xsd:attribute name="elem_id" type="xsd:integer" use="required"/>
          </xsd:extension>
        </xsd:simpleContent>
100   </xsd:complexType>

      <xsd:complexType name="argsType">
        <xsd:sequence>
          <xsd:element name="arg" type="argType" minOccurs="0" maxOccurs="
              unbounded"/>
105     </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType name="argType">
        <xsd:simpleContent>
110       <xsd:extension base="xsd:string">
            <xsd:attribute name="id" type="xsd:integer" use="required"/>
            <xsd:attribute name="direction" type="dirType" use="required"/>
            <xsd:attribute name="type" type="xsd:string" use="required"/>
          </xsd:extension>
115     </xsd:simpleContent>
      </xsd:complexType>

      <xsd:simpleType name="dirType">
        <xsd:restriction base="xsd:string">
120       <xsd:enumeration value="in"/>
          <xsd:enumeration value="out"/>
          <xsd:enumeration value="inout"/>
        </xsd:restriction>
      </xsd:simpleType>
125 </xsd:schema>
```

For each operation provided by one interface, a corresponding *operation* child element should be contained in the *operations* element (lines 72-76). Each operation is characterized by its name and its return type. Moreover, it may have or may have not parameters. So, an *operation* element has two attributes, named *retType* and *name* respectively (line 84-91).

As far as the parameters of an operation are concerned, an *args* element must exist inside each *operation* element. If one operation has parameters, then for each parameter a corresponding *arg* element will be inside the *args* element (lines 102-106). Parameters are characterized by their name, data type and direction mode. Actually, each *arg* element has the *direction* and *type* attributes, while the string contained between the start and the end tag represents the name of the parameter (lines 108-116). Concerning the direction mode of a parameter, the *direction* attribute can take one of the known values (*in*, *out*, *inout*) according to the declaration in lines 118-125. On the other hand, if an operation has no parameters, then the *args* element will be empty.

All the basic elements mentioned so far have an *id* attribute which takes a numeric value as a unique identifier. This identifier will be used for referencing purposes during both the parsing and code generation processes.

For each element-object which realizes an interface, we have to know its name and its realizing interface. Therefore, the *element* element has an *interf_id* attribute whose value refers to the value of the id of the corresponding *interface* element (lines 47-54). The *element* element has also an *id* attribute, while the string between the start and the end tag is the name of the element-object.

Each *capsule* element has an *id* attribute and an *IPaddress* attribute. The latter depicts the IP address of the server machine hosting the server capsule (lines 56-62). As we mentioned previously, server capsules should maintain references, through a location transparency mechanism, so as the communication between remote objects to be possible. Consequently, we have to include information about these references.

A *references* element, including a number of *reference* elements, should be involved in the *capsule* element (lines 78-82). Each *reference* element represents a reference that the current capsule holds to the corresponding element-object. These references are characterized by a name and by the object they refer to. The *reference* element has an *id* attribute and an *elem_id* attribute, with the latter referring to the id of the corresponding element-object. The string inside the tags of the element is the name of the reference (lines 93-100).

Listing 3.2 gives a possible instance of the aforementioned XML schema. Such a XML document can be the first input component to the framework (figure 3.1), according to our example scenario. The XML schema file *architecture.xsd*, mentioned in the root element of the document, is referred to the previously declared XML schema (listing 3.1) and is used here for the validation of the current XML document. It is implied from this XML document that we have a server capsule (lines 31-38), hosted in a machine with IP address *127.0.0.1*. The capsule contains two elements, named *HPServer_Impl* and *StulexServer_Impl* (lines 27-30), realizing two interfaces named *HPServer_Interf* and *StulexServer_Interf* respectively (lines 4-14 and 15-25), on top of Java *RMI* middleware platform (lines 5, 16). The two interfaces provide an operation named *print()* (lines 7, 18) which is implemented by the elements. The functionality of the operations has to do with a printing application and they accept as a parameter a string with the name of the file to be printed (lines 9, 20). The server capsule uses the names *HP* and *Stulex* respectively (lines 33-36) as references to the corresponding element-objects that can be utilized by other elements in order to locate them (e.g. through the Naming Service) and use their services.

Listing 3.2: XML Document of the Platform Independent Information

```
 1  <?xml version="1.0" ?>
    <architecture type="architecture" xmlns:xsi="http://www.w3.org/2001/
        XMLSchema-instance" xsi:noNamespaceSchemaLocation="architecture.xsd">
      <interfaces>
        <interface id="1">
 5        <mdw_platform>RMI</mdw_platform>
          <name>HPServer_Interf</name>
          <operations>
            <operation id="1" retType="void" name="print">
              <args>
10              <arg id="1" direction="in" type="string">filename</arg>
              </args>
            </operation>
          </operations>
        </interface>
15      <interface id="2">
          <mdw_platform>RMI</mdw_platform>
          <name>StulexServer_Interf</name>
          <operations>
            <operation id="1" retType="void" name="print">
20            <args>
                <arg id="1" direction="in" type="string">filename</arg>
              </args>
            </operation>
          </operations>
25      </interface>
      </interfaces>
      <elements>
        <element id="1" interf_id="1">HPServer_Impl</element>
        <element id="2" interf_id="2">StulexServer_Impl</element>
30    </elements>
      <capsules>
        <capsule id="1" IPaddress="127.0.0.1">
          <references>
            <reference id="1" elem_id="1">HP</reference>
35          <reference id="2" elem_id="2">Stulex</reference>
          </references>
        </capsule>
      </capsules>
    </architecture>
```

3.4 Platform Specific Patterns

We have already stated that the basic requirement for our framework is that it does not depend on the middleware platforms assumed by the clients neither on the middleware platforms assumed by the servers, existing in a distributed environment. To achieve the previous, the Illusion Maker generates:

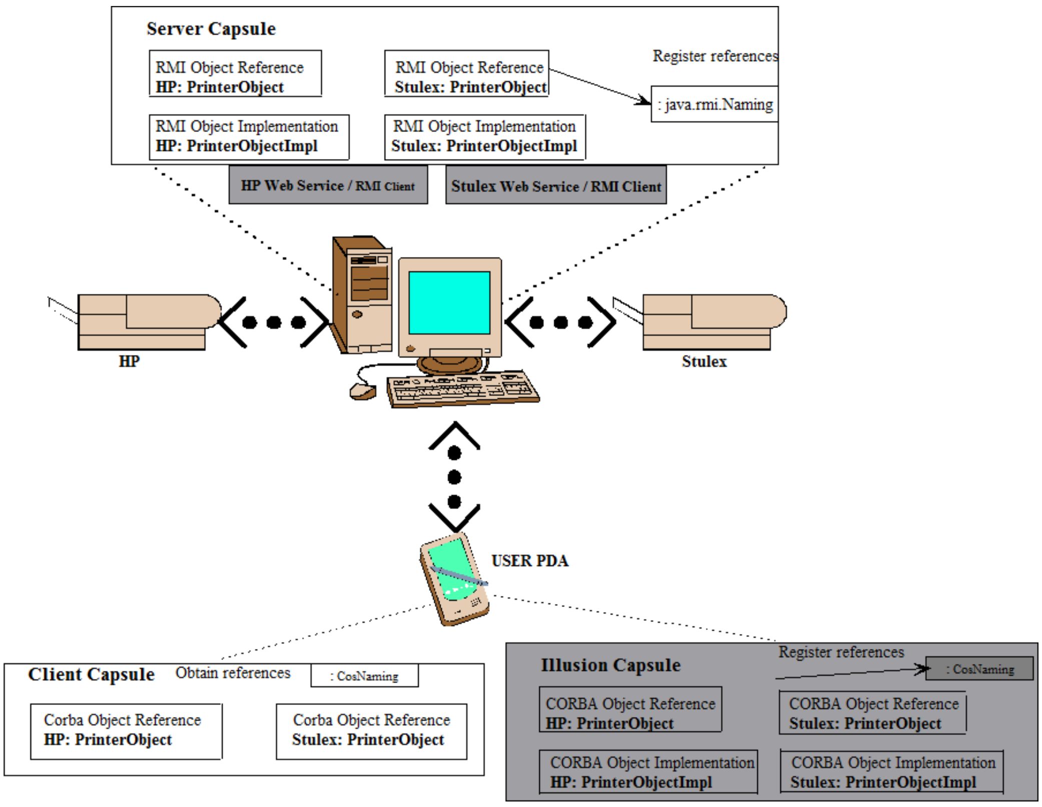
1. Web services that wrap the server objects by providing identical interfaces with the ones provided by the actual objects and play the role of clients for the server objects.
2. Illusion server capsules on top of the middleware infrastructure assumed by the client, which create objects that correspond to the objects required by the client, i.e. they provide the specific interface required by the client applications.
3. The implementation of these objects which play the role of a Web service client to the Web services that wrap the actual server objects.

The generation of all of these software components is based on a number of platform specific patterns, which are given as input to the framework (see figure 3.1). These patterns specify the set of rules that model the descriptions of mappings between the various middleware standards. These rules must contain the following:

- Mapping the build in data types of each middleware platform (such as CORBA and Java RMI) to WSDL, and vice versa.
- Mapping the interface definitions of each middleware platform to WSDL, and vice versa.
- Platform independent parameters which reflect the basic architectural features of a legacy server and whose values can be retrieved from the first input component of the framework (see section 3.3).

In this way, we achieve the transformation of descriptions that are valid in the domain of one middleware infrastructure to identical descriptions that another middleware infrastructure assumes. Roughly, for a pair of platforms A and B assumed respectively by a client and a server, the Illusion Maker requires the existence of two corresponding patterns A-to-WebServices and WebServices-to-B. The patterns specify generically how to generate:

1. The A-specific illusion capsule that hosts the objects invoked by the client application (pattern A-to-WebServices).
2. The A-specific implementations of the objects that serve as Web service clients (pattern A-to-WebServices).



Server Capsule
RMI Object Reference HP: PrinterObject
RMI Object Reference Stulex: PrinterObject
Register references
: java.rmi.Naming
RMI Object Implementation HP: PrinterObjectImpl
RMI Object Implementation Stulex: PrinterObjectImpl
HP Web Service / RMI Client
Stulex Web Service / RMI Client
HP
Stulex
USER PDA
Client Capsule
Obtain references
: CosNaming
Corba Object Reference HP: PrinterObject
Corba Object Reference Stulex: PrinterObject
Illusion Capsule
Register references
: CosNaming
CORBA Object Reference HP: PrinterObject
CORBA Object Reference Stulex: PrinterObject
CORBA Object Implementation HP: PrinterObjectImpl
CORBA Object Implementation Stulex: PrinterObjectImpl

Figure 3.3: The Software Elements generated by Illusion Maker

3. The Web services that serve as B-specific clients to the objects included in the server capsule (pattern WebServices-to-B).

The framework specializes the rules into the source code which leads to the interoperability. Based on the registered information in the first input component and on the set of specified rules in the pattern A-to-WebServices, the framework generates the A-specific interface definitions according to the interfaces of the objects required by the client. In addition, the framework generates the implementations of the objects, which are Web service clients, that the client invokes and the client-side illusion capsule, hosting the aforementioned created objects. Similarly, pattern WebServices-to-B, in conjunction with the registered information about the architecture of the server, is specialized into WSDL descriptions according to the interfaces of the objects offered by the server. Thus, the Web service wrappers for the actual server objects are created. The software elements contained in the grey boxes of figure 3.3 are the ones generated by the Illusion Maker, regarding our case study scenario.

3.4.1 Generation of Platform Specific Illusion Interfaces

To specify patterns, we rely again on XML constructs. Each XML tag, with its nested content inside a platform specific pattern, constitutes a particular rule for the Illusion

Maker. Listing 3.3 gives part of the CORBA-to-WebServices pattern which can be used in our example scenario to allow the client to access the "HP" and the "Stulex" objects through CORBA. This part specifies the rules for generating CORBA IDL interfaces and its structure reflects the structure of a CORBA IDL definition (see listing 1.5). If we examine listing 3.3, we discover the following pattern rules:

- The *interf_exp* rule, containing *literal_exp* rules, the *interf_name* platform independent parameter and an *operation_exp* rule.
- The *operation_exp* rule contains further *literal_exp* rules, the *retType* and *opName* parameters and an *arg_exp* rule.
- The *arg_exp* rule contains a *direction_type_argName* platform independent parameter and a *literal_exp* rule.

The basic rule for the this part of the CORBA-to-WebServices pattern is the *interf_exp* rule (line 18). According to this rule, the Illusion Maker is instructed to generate an IDL interface for every *interface* element in the XML document of the platform independent information. The nested rules within the *interf_exp* start and end tags (lines 19-35) instruct the Illusion Maker to generate the content of an IDL definition. The text involved in *literal_exp* elements reflects the parts of the source code, which are platform specific and are going to be reproduced identically. The platform independent parameters (such as *interf_name*, *opName*, etc.) reflect the architectural features of the legacy server application; their values are retrieved, during the code generation process, from the particular XML document.

The first line of the interface declaration consists of the string *interface*, contained in the first *literal_exp* element (line 19), followed by the name of the interface which is represented by the *interf_name* platform independent parameter (line 20). Following that, the pattern instructs the Illusion Maker to iterate through the operations of the interface; this is realized by the *operation_exp* rule (line 22). For each *operation element* in the XML document of the platform independent information, a corresponding operation declaration should exist inside the interface declaration. An operation declaration consists of the return data type (*retType* parameter, line 24), the name of the operation (*opName* parameter, line 25) and a list of arguments. This task involves another iteration through the list of parameters of each operation (reflected by *arg_exp* rule, line 27). Thus, for each *arg* element in the XML document of the platform independent information, an arg declaration (consisting of the direction mode, the data type and the name of each parameter) is added to the corresponding operation declaration.

Listing 3.3: Generation of CORBA IDL Illusion Interfaces

```
<?xml version="1.0"?>
<pattern type="Interf" platform="Corba">
  <datatypes>
    <datatype id="0" name="void">void</datatype>
    <datatype id="1" name="boolean">boolean</datatype>
    <datatype id="2" name="char">char</datatype>
    <datatype id="3" name="string">string</datatype>
    <datatype id="4" name="byte">octet</datatype>
    <datatype id="5" name="short">short</datatype>
    <datatype id="6" name="unsigned short">unsigned short</datatype>
    <datatype id="7" name="int">long</datatype>
    <datatype id="8" name="unsigned int">unsigned long</datatype>
    <datatype id="9" name="long">long long</datatype>
    <datatype id="10" name="unsigned long">unsigned long long</datatype>
    <datatype id="11" name="float">float</datatype>
    <datatype id="12" name="double">double</datatype>
  </datatypes>
  <interf_exp>
    <literal_exp>interface </literal_exp>
    <interf_name/>
    <literal_exp> { </literal_exp>
    <operation_exp>
      <literal_exp>_newline</literal_exp><literal_exp>_tab</literal_exp>
      <retType/>
      <opName/>
      <literal_exp>(</literal_exp>
      <arg_exp>
        <direction_type_argName/>
        <literal_exp>,</literal_exp>
      </arg_exp>
      <literal_exp>);</literal_exp>
    </operation_exp>
    <literal_exp>_newline</literal_exp>
    <literal_exp>};</literal_exp>
    <literal_exp>_newline</literal_exp>
  </interf_exp>
</pattern>
```

3.4.2 Generation of Platform Specific Object Implementations

The rest of the patterns work in a similar manner. Listing 3.4 is the part of the CORBA-to-WebServices pattern that details the generation of CORBA-specific object implementations, acting as Web service clients (see listing 1.8). This part of the platform specific pattern contains the following rules:

- The *element_exp* rule, which contains several *literal_exp* rules, the *interf_name* parameter, the *element_name* parameter and an *operation_exp* rule.

- The *operation_exp* rule is similar to the corresponding rule described in the previous subsection. It further contains one *addparameter_exp* rule, an *invoke* rule and a *retType_exp* rule.

- The *addparameter_exp* rule contains the *arg_id*, *type* and *direction* parameters.

- The *invoke* rule contains the *retType_exp* rule twice and an *arg_exp* rule.

In particular, the basic *element_exp* rule instructs the Illusion Maker to iterate through the element-objects, which realize each interface (line 26). For each *element* element in the XML document of the platform independent information, a *element_name* Java class is created (lines 40-44). The *operation_exp* rule (line 55) instructs the Illusion Maker to iterate through the list of operations, implemented by the corresponding element. Each operation of this class constructs dynamically a Web service call that accesses a Web service wrapper (lines 55-115). For each parameter of each operation, the *addparameter_exp* rule (lines 72-80) instructs the Illusion Maker to generate a source code line that adds the corresponding parameter to the dynamic operation invocation (i.e. line 22 in listing 1.8). The *invoke* rule with its nested rules (lines 84-101) generate the source code line which realizes the dynamic invocation (i.e. line 24 in listing 1.8). The last *retType_exp* rule (lines 107-110) generates the return type line of a non-void operation (in listing 1.8 the implemented operation is void).

Listing 3.4: Generation of CORBA-specific Object Implementations

```
1  <?xml version="1.0"?>
   <pattern type="ObjectImpl" platform="Corba">
     <datatypes>
       <datatype id="0" name="void" xmltype="AXIS_VOID">void</datatype>
5      <datatype id="1" name="boolean" retValue="false" xmltype="XSD_BOOLEAN"
           javalangtype="Boolean" holder="BooleanHolder">boolean</datatype>
       <datatype id="2" name="char" retValue="'\u0000'" xmltype="XSD_STRING"
           javalangtype="Character" holder="CharHolder">char</datatype>
       <datatype id="3" name="string" retValue="null" xmltype="XSD_STRING"
           javalangtype="String" holder="StringHolder">String</datatype>
       <datatype id="4" name="byte" retValue="-1" xmltype="XSD_BYTE"
           javalangtype="Byte" holder="ByteHolder">byte</datatype>
       <datatype id="5" name="unsigned byte" retValue="-1" xmltype="
           XSD_UNSIGNEDBYTE" javalangtype="Short" holder="ShortHolder">short</
           datatype>
10     <datatype id="6" name="short" retValue="-1" xmltype="XSD_SHORT"
           javalangtype="Short" holder="ShortHolder">short</datatype>
       <datatype id="7" name="unsigned short" retValue="-1" xmltype="
           XSD_UNSIGNEDSHORT" javalangtype="Short" holder="ShortHolder">short</
           datatype>
       <datatype id="8" name="int" retValue="-1" xmltype="XSD_INT"
           javalangtype="Integer" holder="IntHolder">int</datatype>
```

```
    <datatype id="9" name="unsigned int" retValue="-1" xmltype="
        XSD_UNSIGNEDINT" javalangtype="Integer" holder="IntHolder">int</
        datatype>
    <datatype id="10" name="long" retValue="-1" xmltype="XSD_LONG"
        javalangtype="Long" holder="LongHolder">long</datatype>
    <datatype id="11" name="unsigned long" retValue="-1" xmltype="
        XSD_UNSIGNEDLONG" javalangtype="Long" holder="LongHolder">long</
        datatype>
    <datatype id="12" name="float" retValue="-1.0" xmltype="XSD_FLOAT"
        javalangtype="Float" holder="FloatHolder">float</datatype>
    <datatype id="13" name="double" retValue="-1.0" xmltype="XSD_DOUBLE"
        javalangtype="Double" holder="DoubleHolder">double</datatype>
    <datatype id="14" name="BigInt" retValue="java.math.BigInteger.ZERO"
        xmltype="XSD_INTEGER" javalangtype="java.math.BigInteger" holder="
        ObjectHolder">java.math.BigInteger</datatype>
    <datatype id="15" name="decimal" retValue="new java.math.BigDecimal(
        java.math.BigInteger.ZERO)" xmltype="XSD_DECIMAL" javalangtype="java
        .math.BigDecimal" holder="ObjectHolder">java.math.BigDecimal</
        datatype>
  </datatypes>
  <parametermodes>
    <mode name="in">IN</mode>
    <mode name="out">OUT</mode>
    <mode name="inout">INOUT</mode>
  </parametermodes>
  <element_exp>
    <literal_exp>import org.omg.CosNaming.*;</literal_exp>
    <literal_exp>import org.omg.CosNaming.NamingContextPackage.*;</
        literal_exp>
    <literal_exp>import org.omg.CORBA.*;</literal_exp>
    <literal_exp>import org.omg.PortableServer.*;</literal_exp>
    <literal_exp>import org.omg.PortableServer.POA;</literal_exp>
    <literal_exp>import org.apache.axis.client.Call;</literal_exp>
    <literal_exp>import org.apache.axis.client.Service;</literal_exp>
    <literal_exp>import org.apache.axis.encoding.XMLType;</literal_exp>
    <literal_exp>import org.apache.axis.utils.Options;</literal_exp>
    <literal_exp>import javax.xml.rpc.ParameterMode;</literal_exp>
    <literal_exp>import java.util.Properties;</literal_exp>
    <literal_exp>import java.net.*;</literal_exp>
    <literal_exp>import java.lang.*;</literal_exp>
    <literal_exp>public class </literal_exp>
    <element_name/>
    <literal_exp> extends </literal_exp>
    <interf_name/>
    <literal_exp>POA { </literal_exp>
    <literal_exp>private ORB orb;</literal_exp>
    <literal_exp>private String endpoint;</literal_exp>
    <literal_exp>public </literal_exp>
    <element_name/>
    <literal_exp>(String servURL) {</literal_exp>
    <literal_exp>endpoint = servURL;</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>public void setORB(ORB orb_val) {</literal_exp>
    <literal_exp>this.orb = orb_val;</literal_exp>
    <literal_exp>}</literal_exp>
    <operation_exp>
      <literal_exp>public </literal_exp>
      <retType/>
      <opName/>
      <literal_exp>(</literal_exp>
      <arg_exp>
        <direction_type_argName/>
        <literal_exp>,</literal_exp>
      </arg_exp>
      <literal_exp>) {</literal_exp>
      <literal_exp>try {</literal_exp>
      <literal_exp>Service service = new Service();</literal_exp>
      <literal_exp>Call call = (Call) service.createCall();</literal_exp>
      <literal_exp>call.setTargetEndpointAddress(new URL(endpoint));</
          literal_exp>
      <literal_exp>call.setOperationName("</literal_exp>
      <opName/>
      <literal_exp>");</literal_exp>
      <addparameter_exp>
        <literal_exp>call.addParameter("arg</literal_exp>
        <arg_id/>
        <literal_exp>", XMLType.</literal_exp>
        <type/>
        <literal_exp>, ParameterMode.</literal_exp>
        <direction/>
        <literal_exp>);</literal_exp>
      </addparameter_exp>
      <literal_exp>call.setReturnType(XMLType.</literal_exp>
      <retType/>
      <literal_exp>);</literal_exp>
      <invoke>
        <retType_exp id="1">
          <retType/>
          <literal_exp> ret = (</literal_exp>
          <retType/>
          <literal_exp>) </literal_exp>
        </retType_exp>
        <literal_exp>call.invoke(new java.lang.Object[] { </literal_exp>
        <arg_exp>
          <direction_type_argName/>
          <literal_exp>,</literal_exp>
        </arg_exp>
        <literal_exp> } );</literal_exp>
        <retType_exp id="2">
          <literal_exp>_tab</literal_exp>
          <literal_exp>return ret;</literal_exp>
        </retType_exp>
      </invoke>
      <literal_exp>} </literal_exp>
      <literal_exp>catch(Exception ex) {</literal_exp>
      <literal_exp>System.err.println("ERROR: " + ex);</literal_exp>
      <literal_exp>ex.printStackTrace(System.out);</literal_exp>
      <literal_exp>System.out.println("Cannot establish connection with Web
          Service!");</literal_exp>
      <retType_exp id="3">
        <literal_exp>return </literal_exp>
        <retValue/>
        <literal_exp>;</literal_exp>
      </retType_exp>
      <literal_exp>}</literal_exp>
      <literal_exp>}</literal_exp>
    </operation_exp>
    <literal_exp>}</literal_exp>
  </element_exp>
</pattern>
```

3.4.3 Generation of Platform Specific Illusion Capsules

Listing 3.5 is the part of the CORBA-to-WebServices pattern that details how to generate CORBA-specific illusion capsules (see listing 1.7). The basic pattern rule is the *reference_exp* rule (line 3). For every object within a capsule, this rule instructs the Illusion Maker to generate a Java class whose name will be *reference_nameServer* (lines 10-12); *reference_name* (line 11) is the platform independent parameter that denotes the name of a reference with which the corresponding object is associated. The code for this class is given within the *reference_exp* tags. The main method of each class initializes the ORB broker and several other CORBA specific objects, such as the RootPOA and the NameService (lines 13-19). Then, a new instance of the CORBA-specific object implementation is created (*element_name* parameter, lines 20-22), which involves specifying the URI where the corresponding Web service wrapper can be found. The latter is reflected by the *jws_exp* rule (lines 23-39). Finally, the Illusion Maker is instructed to generate code (lines 40-49) that registers the just previously created instance, with the corresponding assumed name (represented by the *reference_name* platform independent parameter, line 47), to the CORBA Naming Service.

Listing 3.5: Generation of CORBA-specific Illusion Capsules

```
<?xml version="1.0"?>
<pattern type="Capsule" platform="Corba">
  <reference_exp>
    <literal_exp>import org.omg.CosNaming.*;</literal_exp>
    <literal_exp>import org.omg.CosNaming.NamingContextPackage.*;</
        literal_exp>
    <literal_exp>import org.omg.CORBA.*;</literal_exp>
    <literal_exp>import org.omg.PortableServer.*;</literal_exp>
    <literal_exp>import org.omg.PortableServer.POA;</literal_exp>
    <literal_exp>import java.net.*;</literal_exp>
    <literal_exp>public class </literal_exp>
    <reference_name/>
    <literal_exp>Server {</literal_exp>
    <literal_exp>public static void main(String args[]) {</literal_exp>
    <literal_exp>try {</literal_exp>
    <literal_exp>ORB orb = ORB.init(args, null);</literal_exp>
    <literal_exp>POA rootpoa = POAHelper.narrow(orb.
        resolve_initial_references("RootPOA"));</literal_exp>
    <literal_exp>rootpoa.the_POAManager().activate();</literal_exp>
    <literal_exp>org.omg.CORBA.Object objRef = orb.
        resolve_initial_references("NameService");</literal_exp>
    <literal_exp>NamingContextExt ncRef = NamingContextExtHelper.narrow(
        objRef);</literal_exp>
    <element_name/>
    <literal_exp> impl = new </literal_exp>
    <element_name/>
    <literal_exp>("http://</literal_exp>
    <IPAddress/>
    <literal_exp>:8080/axis/</literal_exp>
    <jws_exp>
      <jws id="1">
        <literal_exp>WS_</literal_exp>
        <reference_name/>
        <literal_exp>_</literal_exp>
        <server_platform/>
        <literal_exp>Client</literal_exp>
      </jws>
      <jws id="2">
        <reference_name/>
        <literal_exp>Server</literal_exp>
      </jws>
    </jws_exp>
    <literal_exp>.jws");</literal_exp>
    <literal_exp>impl.setORB(orb);</literal_exp>
    <literal_exp>org.omg.CORBA.Object ref = rootpoa.servant_to_reference(
        impl);</literal_exp>
    <interf_name/>
    <literal_exp> href = </literal_exp>
    <interf_name/>
    <literal_exp>Helper.narrow(ref);</literal_exp>
    <literal_exp>NameComponent path[] = ncRef.to_name("</literal_exp>
    <reference_name/>
    <literal_exp>");</literal_exp>
    <literal_exp>ncRef.rebind(path, href);</literal_exp>
    <literal_exp>System.out.println("Corba </literal_exp>
    <reference_name/>
    <literal_exp> started on IP " + InetAddress.getLocalHost().
        getHostAddress()+"\n" );</literal_exp>
    <literal_exp>orb.run();</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>catch(Exception ex) {</literal_exp>
    <literal_exp>System.err.println("ERROR: " + ex);</literal_exp>
    <literal_exp>ex.printStackTrace(System.out);</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>}</literal_exp><literal_exp>_newline</literal_exp>
    <literal_exp>}</literal_exp>
  </reference_exp>
</pattern>
```

3.4.4 Generation of Web Service Wrappers

Finally, listing 3.6 constitutes the WebServices-to-RMI pattern which can be used in our scenario in order to generate the Web service wrappers (see listing 1.9). For each actual remote object, a Web service is created that wraps the functionality of the former and works as a frontend for it (*wrapper_exp* rule, line 21). The generated Web service Java class is named *WS_referenceName_serverplatformClient* as mentioned in lines 24-27 (in our example the two classes will be named *WS_HP_RMIClient* and *WS_Stulex_RMIClient*). The *wrapperoperation_exp* rule (line 30) instructs the Illusion Maker to iterate through the list of operations provided by the server object. For every one of these operations, the framework generates a corresponding web service operation, which is a Java RMI Client. A reference to the Java RMI object is looked up within the RMI registry, using a particular name which has been associated with the object (lines 40-47). By obtaining this reference, the Web service invokes the actual operation provided by the interface of the remote object (*invoke* rule, lines 51-70).

Listing 3.6: Generation of Web Services that serve as Java RMI Clients

```
1  <?xml version="1.0"?>
   <pattern type="ObjectImpl" platform="WS">
     <datatypes>
       <datatype id="0" name="void" retValue="null">void</datatype>
5      <datatype id="1" name="boolean" retValue="false" holder="javax.xml.rpc.
           holders.BooleanHolder">boolean</datatype>
       <datatype id="2" name="char" retValue="'\u0000'" holder="javax.xml.rpc.
           holders.StringHolder">char</datatype>
       <datatype id="3" name="string" retValue="null" holder="javax.xml.rpc.
           holders.StringHolder">String</datatype>
```

```
    <datatype id="4" name="byte" retValue="-1" holder="javax.xml.rpc.holders.ByteHolder">byte</datatype>
    <datatype id="5" name="unsigned byte" retValue="-1" holder="javax.xml.rpc.holders.ShortHolder">short</datatype>
    <datatype id="6" name="short" retValue="-1" holder="javax.xml.rpc.holders.ShortHolder">short</datatype>
    <datatype id="7" name="unsigned short" retValue="-1" holder="javax.xml.rpc.holders.ShortHolder">short</datatype>
    <datatype id="8" name="int" retValue="-1" holder="javax.xml.rpc.holders.IntHolder">int</datatype>
    <datatype id="9" name="unsigned int" retValue="-1" holder="javax.xml.rpc.holders.IntHolder">int</datatype>
    <datatype id="10" name="long" retValue="-1" holder="javax.xml.rpc.holders.LongHolder">long</datatype>
    <datatype id="11" name="unsigned long" retValue="-1" holder="javax.xml.rpc.holders.LongHolder">long</datatype>
    <datatype id="12" name="float" retValue="-1.0" holder="javax.xml.rpc.holders.FloatHolder">float</datatype>
    <datatype id="13" name="double" retValue="-1.0" holder="javax.xml.rpc.holders.DoubleHolder">double</datatype>
    <datatype id="14" name="BigInt" retValue="java.math.BigInteger.ZERO" holder="javax.xml.rpc.holders.BigIntegerHolder">java.math.BigInteger</datatype>
    <datatype id="15" name="decimal" retValue="new java.math.BigDecimal(java.math.BigInteger.ZERO)" holder="javax.xml.rpc.holders.BigDecimalHolder">java.math.BigDecimal</datatype>
  </datatypes>
  <wrapper_exp>
    <literal_exp>import java.rmi.*;</literal_exp>
    <literal_exp>import java.net.*;</literal_exp>
    <literal_exp>public class WS_</literal_exp>
    <reference_name/>
    <literal_exp>_</literal_exp>
    <literal_exp>RMIClient {</literal_exp>
    <interf_name/>
    <literal_exp> objref;</literal_exp>
    <wrapperoperation_exp>
      <literal_exp>public </literal_exp>
      <retType/>
      <opName/>
      <literal_exp>(</literal_exp>
      <arg_exp>
        <direction_type_argName/>
        <literal_exp>,</literal_exp>
      </arg_exp>
      <literal_exp>) {</literal_exp>
      <literal_exp>try {</literal_exp>
      <literal_exp>InetAddress address = InetAddress.getLocalHost();</literal_exp>
      <literal_exp>String serverURL = "rmi://" + address.getHostAddress()
```

```
            + "/</literal_exp>
        <reference_name/>
        <literal_exp>";</literal_exp>
        <literal_exp>objref = (</literal_exp>
        <interf_name/>
        <literal_exp>) Naming.lookup(serverURL);</literal_exp>
        <literal_exp>System.out.println("Obtain a handle on RMI </literal_exp
           >
        <reference_name/>
        <literal_exp> server object");</literal_exp>
        <invoke>
          <retType_exp id="1">
            <retType/>
            <literal_exp> ret = (</literal_exp>
            <retType/>
            <literal_exp>) </literal_exp>
          </retType_exp>
          <literal_exp>objref.</literal_exp>
          <opName/>
          <literal_exp>(</literal_exp>
          <arg_exp>
            <direction_type_argName/>
            <literal_exp>,</literal_exp>
          </arg_exp>
          <literal_exp>);</literal_exp>
          <retType_exp id="2">
            <literal_exp>return ret;</literal_exp>
          </retType_exp>
        </invoke>
        <literal_exp>}</literal_exp>
        <literal_exp>catch(Exception ex) {</literal_exp>
        <literal_exp>ex.printStackTrace();</literal_exp>
        <literal_exp>System.out.println("Cannot establish connection with RMI
            Server!");</literal_exp>
        <literal_exp>System.exit(0);</literal_exp>
        <retType_exp id="3">
          <literal_exp>return </literal_exp>
          <retValue/>
          <literal_exp>;</literal_exp>
        </retType_exp>
        <literal_exp>}</literal_exp>
        <literal_exp>}</literal_exp>
      </wrapperoperation_exp>
      <literal_exp>}</literal_exp>
    </wrapper_exp>
</pattern>
```

This kind of platform specific patterns (*-to-WebServices and WebServices-to-*) can be defined for every possible pair of middleware platforms assumed by a client and a server application respectively. In the Appendix we provide the RMI-to-WebServices and WebServices-to-CORBA patterns, which can be utilized by the Illusion Maker Framework to achieve the interoperability in a scenario where the actual client relies on Java RMI and the legacy server relies on CORBA. In addition, we provide the CORBA-to-RMI and RMI-to-CORBA patterns specifying how to generate CORBA and Java RMI implementations of objects respectively, which access directly the corresponding Java RMI and CORBA actual servers, without using the intermediate Web services.

3.4.5 Mapping of Data Types

At this point, we have to discuss about the *datatypes* and *parametermodes* sections of the platform specific patterns. These sections specify a kind of mapping among the data types and the direction modes of heterogenous middleware platforms. Currently, the framework supports only the existence of build-in data types; it does not support complex data types such as structs, unions etc. Each *datatype* child element inside the *datatypes* element reflects a specific data type. An *id* attribute is used as an identifier for referencing purposes.

For each specific data type, referred in the XML document of the platform independent information (e.g. the return type of an operation, the data type of a parameter), we use a general term. This term is declared within a platform specific pattern as the value of the *name* attribute of the *datatype* element. The string value contained between the start and the end tag is the corresponding name of the data type, which is used by the assumed middleware platform. Here we use Corba IDL data types and Java basic data types.

In addition, we need to know the corresponding Java wrapper classes of the data types[1], the corresponding Holder classes[2], and XML data types[3]. Java wrappers and XML data types are used by the Web service clients in the dynamic object invocation. Holder objects are used when we have to do with parameters whose direction mode is *out* or *inout*. For this information, a corresponding attribute is used inside each *datatype* element (*javalangtype*, *holder* and *xmltype* attributes respectively). We also need to keep a possible return value for each data type, which is going to be returned by an operation whenever something goes wrong, for instance when an exception occurs. This value is declared in the *retValue* attribute. Table 3.1 gives a brief view of the mapping among the data types, assumed as input by the Illusion Maker framework ([2], [3], [6], [13], [14], [15]). Similar is the purpose of the *parametersmodes* section.

[1]java.lang.* classes which wrap a value of the corresponding primitive types in an object (e.g. java.lang.Integer, java.lang.Float etc.).

[2]org.omg.Corba.* and javax.xml.rpc.holders.* classes for Corba IDL[2] and JAX-RPC [15] respectively which use a value attribute to keep the value of an out or inout parameter (e.g. org.omg.CORBA.LongHolder, javax.xml.rpc.holders.ByteHolder etc.).

[3]org.apache.axis.encoding.XMLType.* constants which have been declared as XML type QNames to indicate the corresponding XML data types, e.g. XSD_STRING, XSD_INT etc. (see [3]).

Basic Type	**XML Type**	**Java Type**	**Java Wrapper**	**Java Holder**	**Corba IDL Type**
void	AXIS_VOID	-	-	-	void
boolean	XSD_BOOLEAN	boolean	Boolean	BooleanHolder	boolean
char	XSD_STRING	char	Character	CharHolder	char
string	XSD_STRING	String	String	StringHolder	string
byte	XSD_BYTE	byte	Byte	ByteHolder	octet
unsigned byte	XSD_UNSIGNEDBYTE	short	Short	ShortHolder	short
short	XSD_SHORT	short	Short	ShortHolder	short
unsigned short	XSD_UNSIGNEDSHORT	short	Short	ShortHolder	unsigned short
int	XSD_INT	int	Integer	IntHolder	long
unsigned int	XSD_UNSIGNEDINT	int	Integer	IntHolder	unsigned long
long	XSD_LONG	long	Long	LongHolder	long long
unsigned long	XSD_UNSIGNEDLONG	long	Long	LongHolder	unsigned long long
float	XSD_Float	float	Float	FloatHolder	float
double	XSD_DOUBLE	double	Double	DoubleHolder	double

Table 3.1: Data Types Mapping

Finally, we can see in the *literal_exp* elements of the platform specific patterns some strings such as *_newline* or *_tab*. These are special instructions which tell the Illusion Maker to change a line (the former) or leave a tab space (the latter) while generating the source code and writing it to the output file. For simplification purposes, in the listings shown here, most of these declarations are omitted. This is the general function of the platform specific patterns which is being done by the Illusion Maker in order to refine them into the desirable source code.

3.5 Design and Implementation of the Framework

Figure 3.4 gives a more refined view of the main subsystems that constitute the Illusion Maker. As we see, the framework consists of two parsers. The first parser accepts as input the XML document which contains the platform independent information. This parser converts the XML structure to a corresponding hierarchy of objects, which hold the information contained in the XML document.

The purpose of the second parser is to parse the platform specific pattern and translate it into the desirable source code. The platform specific pattern is given as input to this parser, and the output of the first parsing process feeds its data into the current process. The result is another hierarchy of objects into which the content of the generated source code is organized. Finally, the latter, more refined hierarchy of objects, is forwarded to the File Utility subsystem, which is in charge of storing the source code into suitable

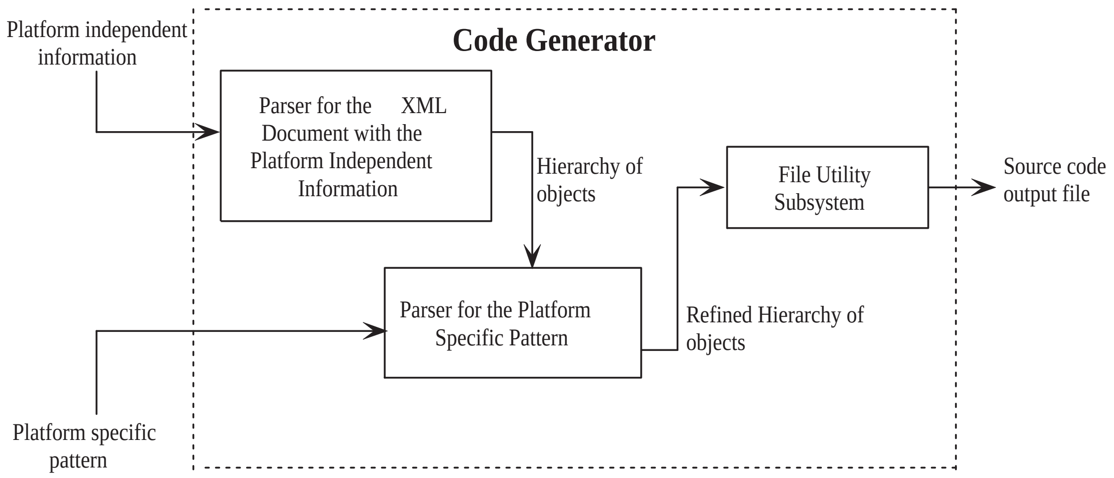

Platform independent information
Code Generator
Parser for the XML Document with the Platform Independent Information
Hierarchy of objects
File Utility Subsystem
Source code output file
Parser for the Platform Specific Pattern
Refined Hierarchy of objects
Platform specific pattern

Figure 3.4: The Subsystems of the Illusion Maker

output files.

In the following subsections, we elaborate the details on the design and the implementation of these subsystems.

3.5.1 Parsing XML Documents

As we have stated above, the basic components of the framework are the two parsers which are responsible for specializing the platform specific patterns into the desirable source code. Since both the platform independent information and the platform specific patterns are stored in XML documents, we need parsers which will read and extract the data contained in the specific XML documents.

For this purpose, we implemented the two parsers on the basis of the *Simple API for XML* (SAX) [10] [12]. *SAX* provides an interface (the *XMLReader* interface) for event-based parsing of XML files. It is a standard that describes how a SAX parser should be written and which events must be supported. The SAX parser does not do anything to the XML data other than trigger certain events. It is up to the user's need and demand to determine what happens when these events take place. So, the user can implement functions for copying the data into a data structure of a native programming language, transforming it into a presentation format or applying a style to it.

The parser reads a document from beginning to end. While doing so, it encounters start tags, end tags, text, comments, processing instructions and more. In SAX, the parser is based on a callback mechanism. This mechanism provides the *ContentHandler* interface, which the client application should implement to receive notification of the document content. The client application will instantiate a specific instance of the *ContentHandler* interface and associate it with the parser that is going to read the document. As the parser reads the document, it tells the client application what it sees as it sees it (e.g. start tags, end tags etc.) by calling back to the methods in the registered *ContentHandler* object. The user-defined implementations of these methods determine what to do to the parsed data [10] [12].

3.5.2 The XML Parsers of the Framework

The Illusion Maker framework reflects a case, where the use of XML parsers is necessary. The first parser is a SAX parser which reads the XML document of the platform independent information. What it does is merely reading the whole document and extracting the contained text data, that is the text within the tags and the attribute values. The result is to copy the data into a list of objects belonging to a class hierarchy that conforms with the structure of the parsed XML document.

The second parser is in charge of parsing the XML document, which specifies the platform specific pattern. We have already mentioned that the platform specific pattern specifies the set of rules, according to which the source code will be generated. Each tag represents a particular instruction for the parser. So, as the parser reads the document, for each pattern rule which is being seen, a particular operation is executed which refines the input. The necessary platform independent information, involved in the server architectural parameters of the rules, is provided to this process, by iterating through the set of lists generated by the first parser. The final output is organized again to a class hierarchy which is ready to be transformed into plain source code.

The UML diagram in figure 3.5 shows the classes and the dependencies between them, which implement the basic functionality of the Illusion Maker framework. In the diagram, the *XMLParser* abstract class is the basic class encapsulating the parsing functionality. Its non-abstract method *parseXML()* implements this functionality, making use of the *XMLReader*[4] and *ContentHandler*[5] interfaces. An instance of the former interface is created with respect to a Xerces SAX Parser[6], which is then attached to a *ContentHandler* instance, using the *DefaultHandler*[7] adapter class. The *xmlfilename* attribute reflects the name of the XML Document to be parsed. The three abstract methods *checkType()*, *isValid()* and *parseXML()* realize the whole parsing operation.

3.5.3 Parsing the Platform Independent Information

The *ArchXMLParser* class specializes the *XMLParser* abstract class, implementing the specific parser for parsing the XML document of the platform independent information.

[4]org.xml.sax.XMLReader: http://java.sun.com/j2se/1.4.2/docs/api/org/xml/sax/XMLReader.html

[5]org.xml.sax.ContentHandler: http://java.sun.com/j2se/1.4.2/docs/api/org/xml/sax/ContentHandler.html

[6]Xerces is the XML Parser from the Apache Software Foundation: http://xml.apache.org

[7]org.xml.sax.helpers.DefaultHandler: http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/helpers/package-summary.html

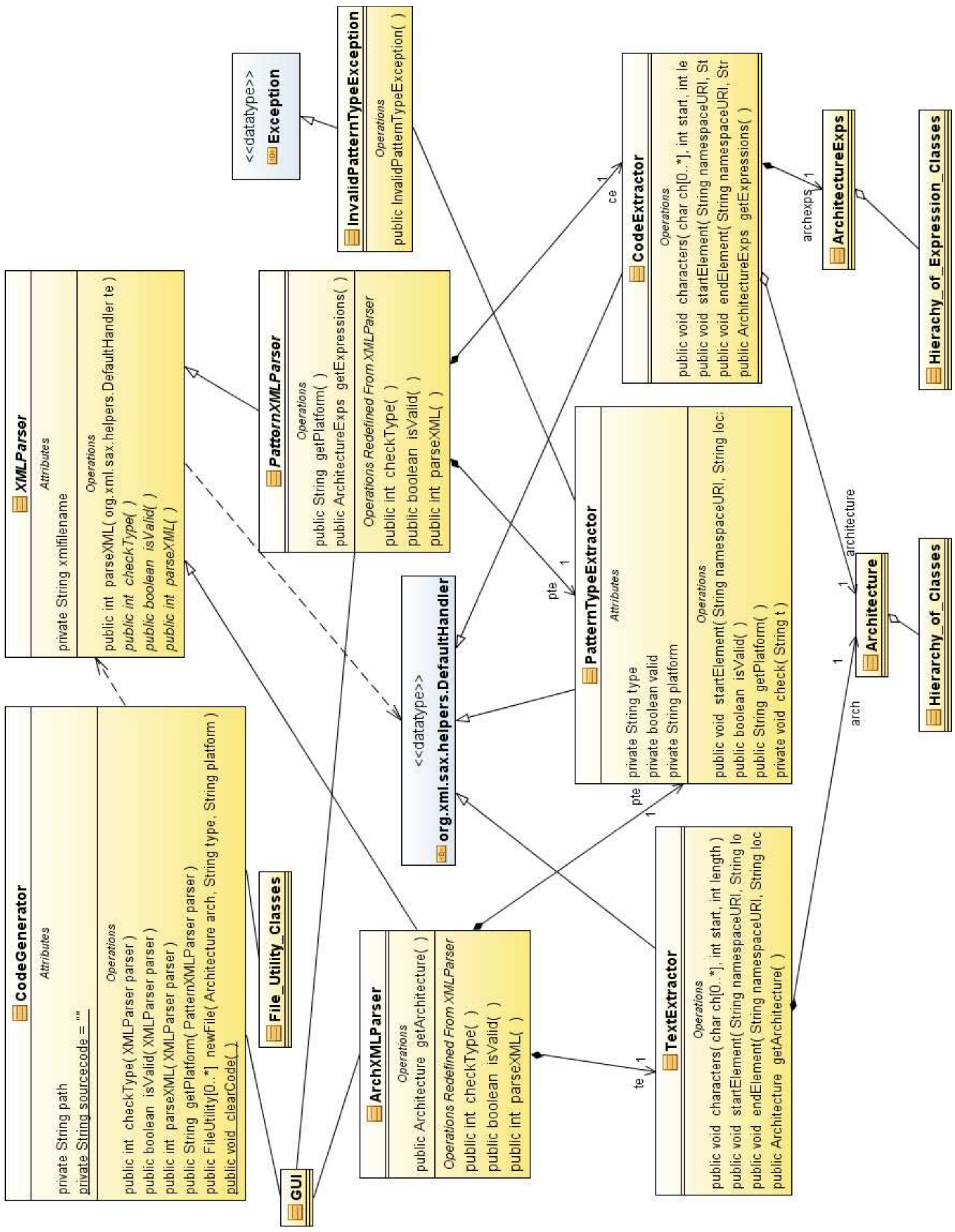

CodeGenerator
Attributes
private String path
private String sourcecode = ""
Operations
public int checkType(XMLParser parser)
public boolean isValid(XMLParser parser)
public int parseXML(XMLParser parser)
public String getPlatform(PatternXMLParser parser)
public FileUtility[0..*] newFile(Architecture arch, String type, String platform)
public void clearCode( )
XMLParser
Attributes
private String xmlfilename
Operations
public int parseXML(org.xml.sax.helpers.DefaultHandler te)
public int checkType( )
public boolean isValid( )
public int parseXML( )
PatternXMLParser
Operations
public String getPlatform( )
public ArchitectureExps getExpressions( )
Operations Redefined From XMLParser
public int checkType( )
public boolean isValid( )
public int parseXML( )
<<datatype>>
Exception
InvalidPatternTypeException
Operations
public InvalidPatternTypeException( )
File_Utility_Classes
GUI
ArchXMLParser
Operations
public Architecture getArchitecture( )
Operations Redefined From XMLParser
public int checkType( )
public boolean isValid( )
public int parseXML( )
<<datatype>>
org.xml.sax.helpers.DefaultHandler
CodeExtractor
Operations
public void characters(char ch[0..*], int start, int le
public void startElement(String namespaceURI, St
public void endElement(String namespaceURI, Str
public ArchitectureExps getExpressions( )
PatternTypeExtractor
Attributes
private String type
private boolean valid
private String platform
Operations
public void startElement(String namespaceURI, String loc
public boolean isValid( )
public String getPlatform( )
private void check(String t)
TextExtractor
Operations
public void characters(char ch[0..*], int start, int length)
public void startElement(String namespaceURI, String lo
public void endElement(String namespaceURI, String loc
public Architecture getArchitecture( )
ArchitectureExps
Hierachy_of_Expression_Classes
Architecture
Hierachy_of_Classes
ce
1
pte
1
pte
1
te
1
archexps
1
architecture
1
arch
1

Figure 3.5: The Classes of the Illusion Maker Design

The *TextExtractor* class realizes the *ContentHandler* interface, and extends the corresponding *DefaultHandler* class. So, *TextExtractor* is the callback mechanism whose methods will be invoked by the parser while reading the XML data. The *PatternTypeExtractor* class is another realization of the *ContentHandler* interface. Its functionality is to check whether the XML document which is going to be parsed is a suitable XML document according to the functionality of the framework.

The *ArchXMLParser* should instantiate the *PatternTypeExtractor* and *TextExtractor* classes. The parsing process consists of the following steps:

1. Using the *PatternTypeExtractor* instance of the callback mechanism, the parser checks the suitability of the XML document to be parsed.

   - The suitability is indicated by the value of the *type* attribute, which should exist in the *architecture* root element of the particular XML document. In this case, this value must be equal to "*architecture*" (listing 3.1, lines 32-36 and listing 3.2, line 2). The operation is implemented by the *checkType()* and *isValid()* redefined methods of the ArchXMLParser class.

2. Using the *TextExtractor* instance of the callback mechanism, the parser reads the XML document and extracts the contained data. During this process, the overridden methods (*characters()*, *startElement()*, *endElement()*) of the TextExtractor class are called back by the parser, building the set of objects which hold the platform independent information. The whole operation is encapsulated in the *parseXML()* redefined method of the ArchXMLParser.

This parsing process results in the creation of an instance of the *Architecture* class, which depicts the corresponding XML structure of the platform independent information. This instance is created by the methods of the callback mechanism. Figure 3.6 gives an overview of this class hierarchy.

We can easily observe that the XML structure has been transformed into a class structure. Each particular XML element, which specifies a basic component of the server architecture, is represented by a corresponding class (e.g. capsule, reference, element, interface, operation, argument). Each set of basic components is represented by a vector-list of the corresponding objects, which is contained in the particular object: a *Capsule* object includes a set of *Reference* objects, an *Interf* object includes a set of *Operation* objects, with each one of them including a set of *Arg* objects. As a whole, an *Architecture* object will contain sets of *Capsule*, *Element* and *Interf* objects. Attributes and child elements of the XML elements that correspond to basic features of the components, are represented by specific class attributes. Consequently, the *Architecture* instance, created during the parsing of the XML document of the platform independent information, is hereafter available to feed its data into the platform specific parsing process.

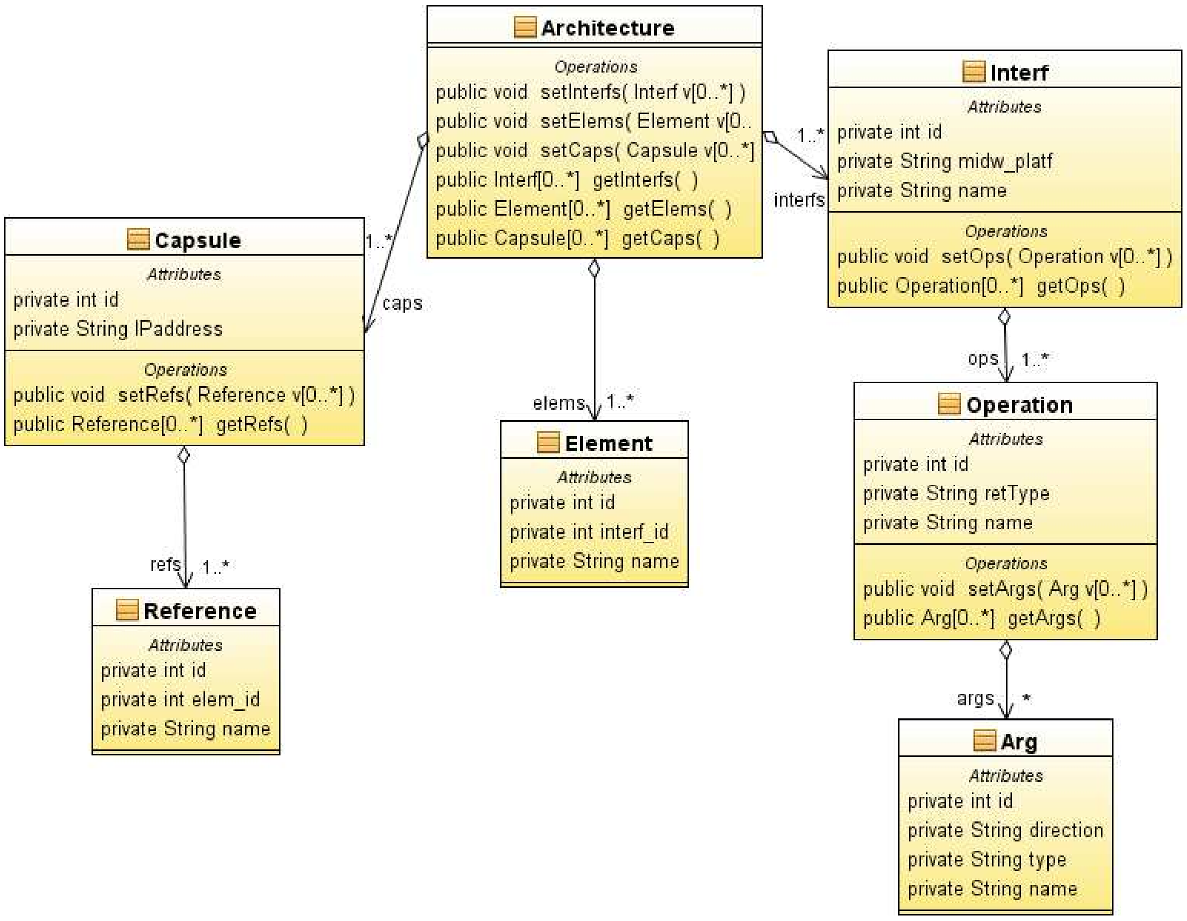


Figure 3.6: The Architecture Class Hierarchy

3.5.4 Parsing the Platform Specific Pattern

As far as the platform specific patterns parsing is concerned, we follow a similar manner with the one described above with respect to the parsing of the platform independent information. The *PatternXMLParser* class (figure 3.5) constitutes the specific parser, specializing the *XMLParser* class, to parse the XML documents which form the platform specific patterns. The *PatternTypeExtractor* realization of the *ContentHandler* interface is used again for the same purpose as previously mentioned, while the *CodeExtractor* provides the callback mechanism which is utilized by the parser as it reads the platform specific pattern. Similarly, the *PatternXMLParser* instantiates the *PatternTypeExtractor* and *CodeExtractor* classes, according to the following steps:

1. Using a *PatternTypeExtractor* instance of the callback mechanism, the parser confirms the suitability of the XML document to be parsed.

    - Here, the permissible values of the *type* attribute of the *pattern* root element, that indicate the suitability of the XML documents, are "*Interf*", "*Capsule*" and "*ObjectImpl*". Each of the previous values correspond to the part of a platform specific pattern, being responsible for the generation of platform specific illusion interfaces (listing 3.3 line 2), illusion capsules (listing 3.5 line 2) and object implementations (listings 3.4 and 3.6 line 2) respectively. This operation is implemented by the *checkType()* and *isValid()* redefined methods of the PatternXMLParser.

2. Provided that the specified XML document is suitable, the parser parses the platform specific pattern. While doing so, for every pattern rule for which it receives notification, it invokes the overriden methods (*characters()*, *startElement()*, *endElement()*) of the *CodeExtractor*. In addition, it retrieves data from the already created *Architecture* instance. During this procedure, for each specific rule seen in the platform specific pattern, a corresponding rule object is created. The PatternXML-Parser *parseXML()* overriden method incorporates the whole operation.

In the wake of this parsing process, an *ArchitectureExps* instance is created which contains a list of rule objects, as detailed in the following subsection. These objects belong to a class hierarchy derived by an abstract class, named *Expression*. Afterwards, the list of *Expression* objects is ready to be converted to plain source code and be written to a proper output file.

3.5.5 Utilizing Interpreter Design Pattern

The platform specific patterns specify the rules which determine generically how to generate the necessary source code in order that the middleware platform interoperability is achieved. Each XML element within a platform specific pattern is a pattern rule. The *PatternXMLParser* produces a list of code generation instructions. To simplify the code generation process, we chose to apply the *Interpreter Design Pattern* [8]. This design pattern describes how to define a representation for the pattern rules.

According to the Interpreter design pattern, we use a class to represent each pattern rule. Therefore, every platform specific pattern can be represented by objects of a class hierarchy, derived by the *Expression* abstract class. The diagram in figure 3.7 gives the whole structure of the *Expression* class hierarchy, which is assumed by the Interpreter design pattern, as applied to our framework.

The *Expression* abstract class provides the interface which the framework uses to interpret the platform specific pattern. It declares an abstract *generate()* method that is common to all Expression subclasses and reflects the functionality of generating the source code. We can see that each rule in the XML document which contains the platform specific pattern (see section 3.4) is represented by a specific Expression subclass. Moreover, the *sourcecode* static string attribute will gradually accumulate the being generated source code.

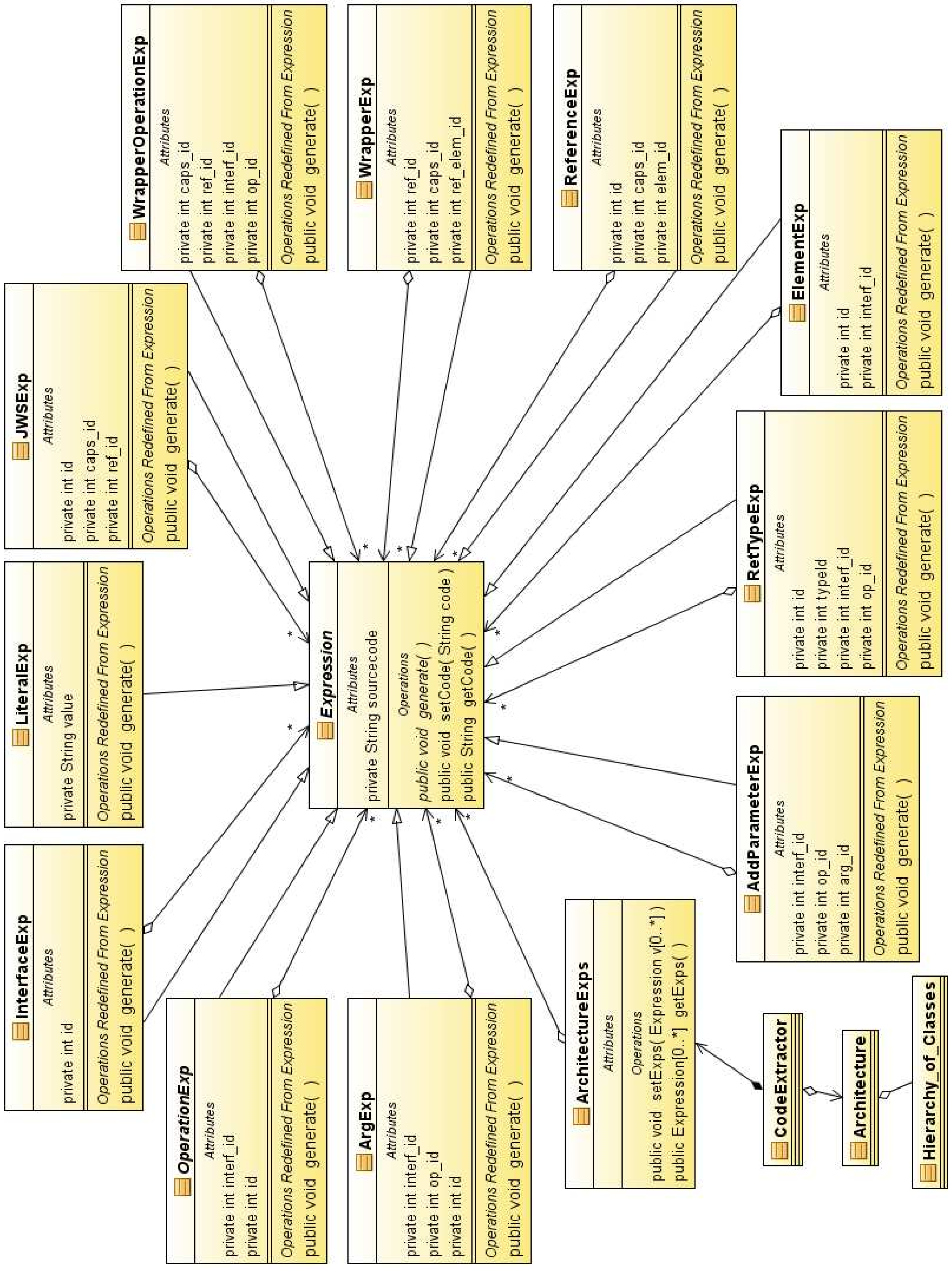


Figure 3.7: Interpreter Design Pattern

In section 3.4 we mentioned that the *literal_exp* XML elements represent the non-parameterized platform specific parts of the source code, which will be reproduced as it is. Furthermore, to refer to the corresponding platform independent information which has to be retrieved, we use some elements, as parameters, inside the platform specific patterns. So, we can consider that these two components of our rules, that is the *platform independent parameters* and the *literal_exp* elements, constitute *terminal expressions.* For every one of these components, a *LiteralExp* instance has to exist in the list of objects, generated by the *PatternXMLParser.* The overriden *generate()* method of these objects appends the corresponding terminal sequence to the currently contained string in the Expression abstract class *sourcecode* attribute.

The rest of the XML elements (e.g. interf_exp, element_exp etc) in a platform specific pattern form the *nonterminal expressions* of the rules. In the generated list of objects, one particular object of a corresponding subclass is required for every corresponding piece of platform independent information. Thus, for each *Interf* instance in the constructed *Architecture* class hierarchy (i.e. for each *interface* element in the platform independent information), an *InterfaceExp* object will be created.

Each object that represents a nonterminal expression contains a list of *Expression* references to other objects that can be terminal and/or nonterminal expressions. The objects included in these lists correspond to the nested rules contained within other rules. As an example, the name of an interface (represented by the *interf_name* parameter) will be included as a LiteralExp object in the list of objects of the corresponding InterfaceExp object. Similarly, for every operation of an interface, the corresponding InterfaceExp object should contain in its list of objects an *OperationExp* instance; the latter contains, in its particular object list, other terminals-LiteralExp instances (e.g. the name and the return type of the operation) and nonterminals (e.g. one *ArgExp* instance for every operation parameter), and so on. The nonterminal classes have also various *id* attributes for referencing purposes. Figure 3.8 gives the hierarchy of objects that represents a CORBA interface definition, arising from the parsing of the corresponding part of CORBA-to-WebServices pattern (see Listing 1.5).

The *generate()* methods of the nonterminal classes implement an iteration through their list of Expression objects. This iteration involves calling recursively the corresponding *generate()* method of each object. Since all nonterminals are finally refined to terminals (e.g. all the nonterminal Expression objects are refined to terminals-LiteralExp objects, see figure 3.8), we end up in calling the LiteralExp *generate()* method. The source code is generated gradually and is accumulated as a string sequence, in the Expression class *sourcecode* static attribute.

The *Architecture* class hierarchy forms the context, containing information which has to be global to the interpreter's environment. Here, this information participates in the platform specific pattern parsing and in the code generation.

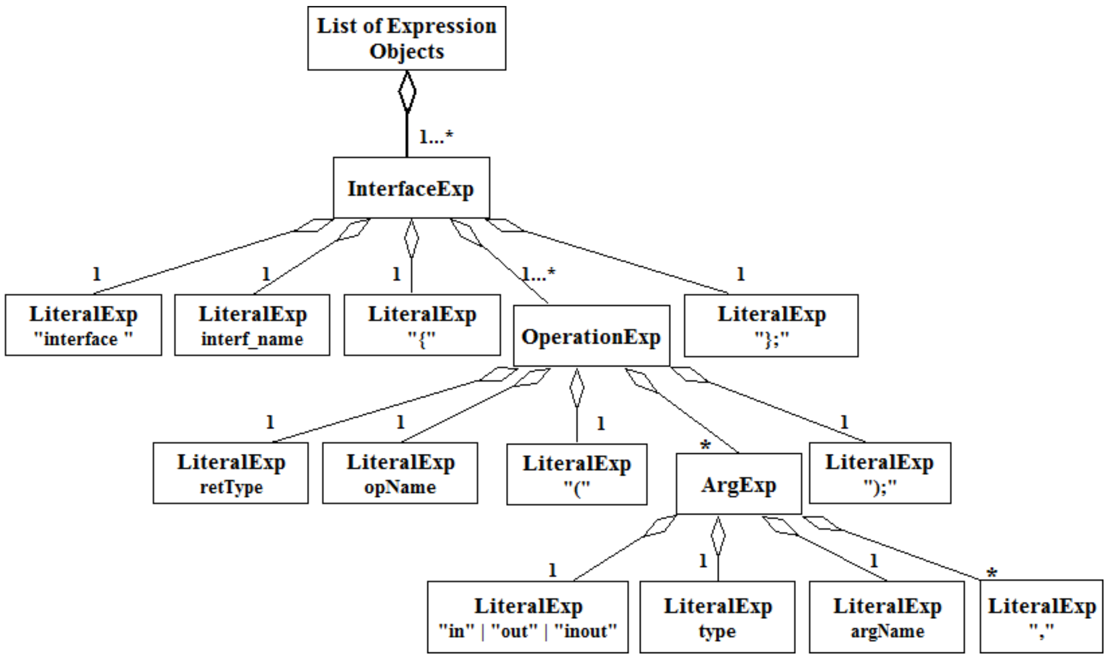


Figure 3.8: List of Expression Objects for a CORBA IDL Interface

This is the logic behind the construction of the platform specific patterns and their parsing. We translated the XML structure of the platform specific patterns into an hierarchy of Expression classes. The Interpreter design pattern provides a simple way to define a grammar, which afterwards can be easily extended. Since classes are used for representing grammar rules, the grammar can be extended using inheritance. Existing expressions can be modified incrementally and new expressions can be defined as variations of old ones. In addition, classes in the abstract syntax tree usually have similar implementations [8].

3.5.6 Mapping of Data Types

The mapping of data types between the different middleware infrastructures is also part of platform specific patterns ([2], [3], [6], [13], [14], [15]). We have already referred to the declaration of basic data types mapping within the platform specific patterns (subsection 3.4.4). During their parsing process, a list of *Type* instances is created. For each *datatype* XML element, a corresponding *Type* object is created. The object has attributes which correspond to the attributes of the *datatype* element. When the parser has to retrieve a specific data type, it iterates through the list of *Type* objects and gets the suitable data type information. Similar is the case with the mapping of parameters direction modes. Figure 3.9 gives the structure of *Type* and *ParameterMode* classes.

3.5.7 Creating the Output Source File

The *CodeGenerator* class (figure 3.5) is the front-end which wraps the functionality of the framework. A user wishing to use the framework should instantiate an instance of this

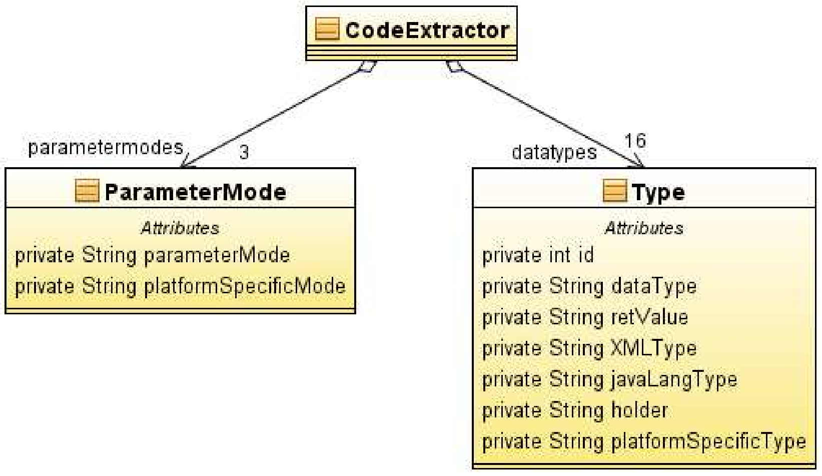


Figure 3.9: Classes for Parameter Direction Modes and Data Types Mapping

class. Moreover, it should instantiate instances of the *ArchXMLParser* and *PatternXML-Parser* classes. The filesystem path where the generated source file will be placed is declared by the *path* string attribute. The three methods, *checkType()*, *isValid()* and *parseXML*, are wrappers of the corresponding XMLParser methods. According to the specific instance of an XMLParser subclass which is given as parameter to every method, the identical method will be invoked in this instance.

As long as the entire parsing process will have been completed successfully, the *ArchitectureExps* instance will be available, containing the constructed list of *Expression* objects. The list of the Expression objects is forwarded to the File Utility subsystem, in order to convert the object hierarchy into a plain source code, based on the Interpreter design pattern (subsection 3.5.5). Figure 3.10 shows the class structure of the File Utility system.

The *FileUtility* class implements operations related to files handling, such as reading or writing data from/to a file, copying files, getting the extension of a file, checking if a file exists, setting and getting the path of the utilizing file, etc. These operations complement the functionality of the framework, as we will see in section 3.7. An instance of this class can be used for writing the source code in a plain form.

The *FileType* abstract class is the interface which provides the functionality for creating the suitable files where the source code is going to be stored. This functionality is reflected by the *newFile()* abstract method. Every *FileType* subclass represents a particular type of

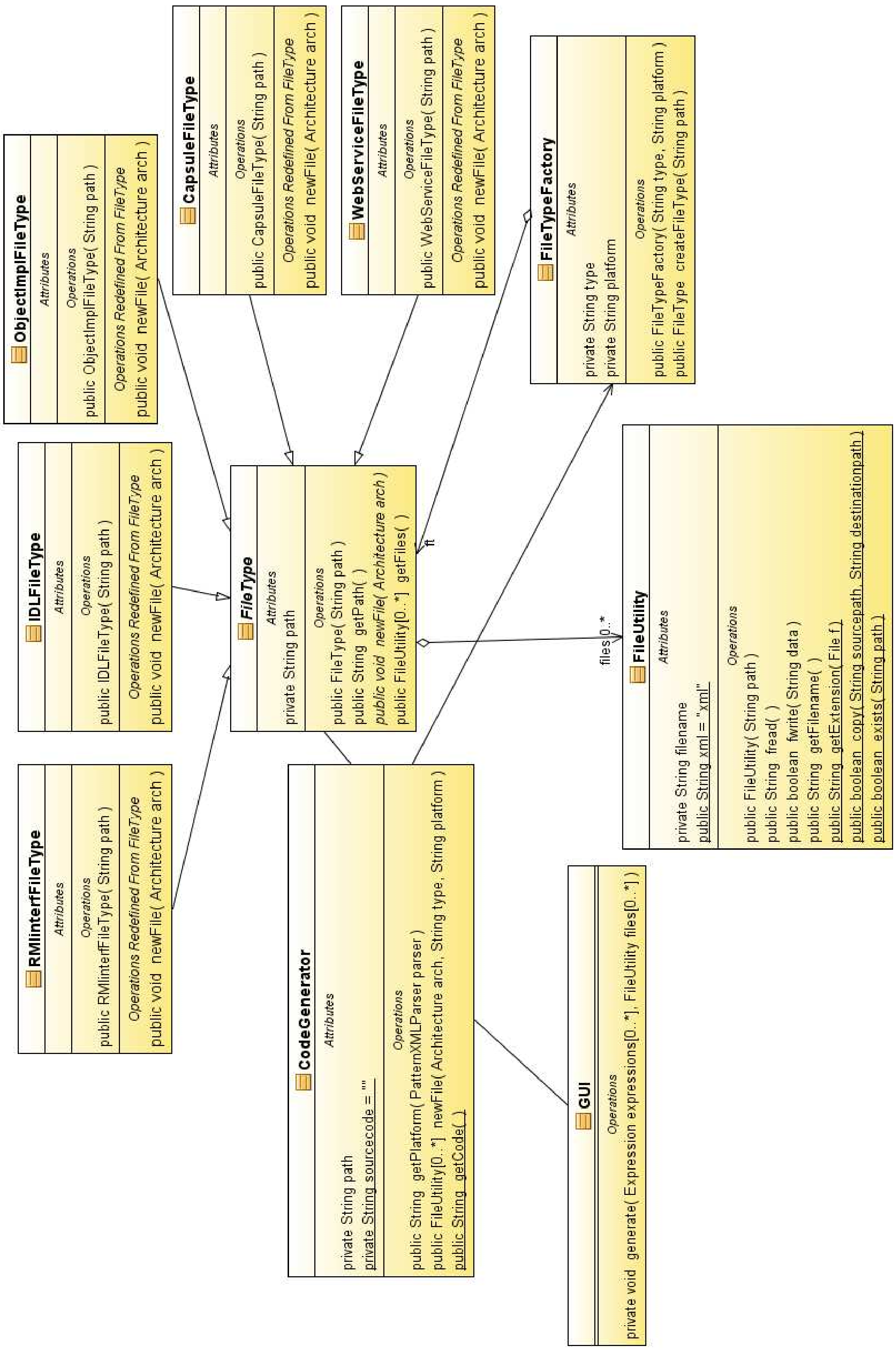

ObjectImplFileType
Attributes
Operations
public ObjectImplFileType(String path)
Operations Redefined From FileType
public void newFile(Architecture arch)
CapsuleFileType
Attributes
Operations
public CapsuleFileType(String path)
Operations Redefined From FileType
public void newFile(Architecture arch)
WebServiceFileType
Attributes
Operations
public WebServiceFileType(String path)
Operations Redefined From FileType
public void newFile(Architecture arch)
FileTypeFactory
Attributes
private String type
private String platform
Operations
public FileTypeFactory(String type, String platform)
public FileType createFileType(String path)
IDLFileType
Attributes
Operations
public IDLFileType(String path)
Operations Redefined From FileType
public void newFile(Architecture arch)
FileType
Attributes
private String path
Operations
public FileType(String path)
public String getPath( )
public void newFile(Architecture arch)
public FileUtility[0..*] getFiles( )
ft
files 0..*
FileUtility
Attributes
private String filename
public String xml = "xml"
Operations
public FileUtility(String path)
public String fread( )
public boolean fwrite(String data)
public String getFilename( )
public String getExtension(File f)
public boolean copy(String sourcepath, String destinationpath)
public boolean exists(String path)
RMIinterfFileType
Attributes
Operations
public RMIinterfFileType(String path)
Operations Redefined From FileType
public void newFile(Architecture arch)
CodeGenerator
Attributes
private String path
private String sourcecode = ""
Operations
public String getPlatform(PatternXMLParser parser)
public FileUtility[0..*] newFile(Architecture arch, String type, String platform)
public String getCode( )
GUI
Operations
private void generate(Expression expressions[0..*], FileUtility files[0..*])

Figure 3.10: The File Utility Subsystem

source file, with their *newFile()* redefined method implementing the operation of creating such a file.

The *CodeGenerator newFile()* method wraps the procedure of creating the files for storing the source code. Based on the platform specific pattern *type* that was parsed and on the *middleware platform* to which the generated source code is related, this method creates a specific instance of a *FileType* subclass. The type of the pattern is indicated by the *type* attribute of the *pattern* root element, while the middleware platform is indicated by the *platform* attribute of the same root element (see listings in section 3.4). The creation of the FileType instance is carried out by a factory class, responsible for creating instances of the FileType subclasses. This factory class is the *FileTypeFactory* class; for instance, if a Web service wrapper is going to be generated, a *WebServiceFileType* instance will be created. Following that, the *newFile()* method is called in this instance. This method uses the *Architecture* instance to retrieve information to construct the name of the file.

In the previous example, for every actual remote object, the method has to create a *.jws* file, containing the source code of a Web service wrapper. Considering our example scenario, two Web services would be created taking the name, according to the pattern *WS_referenceName_serverPlatformClient.jws*, that is WS_HP_RMIClient.jws and WS_Stulex_RMIClient.jws. The *referenceName* and *serverPlatform* data will be retrieved from the *Architecture* instance (see listing 3.2). In particular, the *newFile()* method does not create the actual file, but creates a *FileUtility* instance which is associated with the actual file to be created. The latter instance is added to a list of *FileUtility* instances which the *FileType* main class maintains. This list contains as many FileUtility instances as does the number of the source files, which must be created. In the example mentioned, the list will contain two FileUtility instances.

Provided that the previous operation was successful, the already created *FileUtility* instances are about to store the source code contained in *Expression* instances. Framework *generate()* method (*GUI* class in figure 3.10) iterates through the Expression instances list and invokes the corresponding *generate()* method in each instance. For each instance, the content of the Expression *sourcecode* static attribute, which is the actually being generated source code, is retrieved and appended to the identical CodeGenerator attribute. The content of the latter string attribute is written to the appropriate output file by the FileUtility *fwrite()* method.

After the proper completion of this recurring process, the output source files can be found in the initially specified path of the file system. These source files are about to be utilized making feasible the intercommunication between the heterogenous middleware infrastructures.

3.6 Scalability Issues

The design which was followed provides the framework with a satisfactory degree of scalability. This is useful for future extensions and modifications of its functionality.

The basic feature which contributes to the scalability is our design choice to use Web services as an intermediate platform to facilitate interoperability between different platforms. For a client platform A and a server platform B, an alternative approach would be to generate an A-specific illusion capsule. This capsule contains A-specific objects whose implementations access directly the B-specific actual server objects, instead of their Web service wrappers. That is, the A-specific pattern should specify generically how to generate A-specific implementations of the objects required by the user, which play the role of the client to the B-specific server objects (see the CORBA-to-RMI and RMI-to-CORBA patterns in the appendix).

For every possible pair of platforms A and B, which may be used either by a client or a server within a distributed environment, we would have to specify an A-to-B and a B-to-A pattern. In the same way, for a new platform C, we would have to specify the A-to-C, the C-to-A, the B-to-C and the C-to-B patterns in order to be able to utilize it by means of our framework. Alternatively, with our approach we only have to specify the C-to-WebServices and the WebServices-to-C patterns, in a similar manner with the already existing *-to-WebServices and WebServices-to-* patterns. Therefore, the use of an intermediate platform, such as Web services, simplifies the work to accomplish interoperability; this is clearer when we want to incorporate a new middleware platform.

Next we discuss briefly the procedure which someone has to consider, should he want to extent the functionality of the framework by incorporating a new middleware platform. Suppose X is a new middleware platform released in the market. First of all, the XML Schema shown in listing 3.1 should be slightly changed. Specifically, an extra enumeration value should be included in the set of possible values for the *mdw_platform* element (lines 64-70) of every *interface* element in the XML document of the platform independent information. This is being done in order to maintain the validity of the XML document.

Afterwards, we have to make available the X-to-WebServices and WebServices-to-X patterns. To construct these patterns, we have to consider the facilities that the platform X offers for implementing the distributed applications. We have to examine possible examples and organize the code in a similar manner to the existing patterns. The code should be organized into platform independent and platform specific parts. The structure should rely on the architecture, implied from the *Interpreter* design pattern (figure 3.7). If it is necessary to define new nonterminal expressions for the pattern rules (according to the structure of the patterns implied by the new middleware platform), these rules can easily be added to the existing context, by defining new subclasses, specializing further the *Expression* abstract syntax tree. Such a new subclass will maintain a vector-list of Expression references and will implement an identical *generate()* method.

In addition, we must define a similar data types mapping between the possible data types supported by the new platform and the existing data types. The X-to-WebServices pattern will specify generically how to generate interface definitions, according to the interfaces of the objects required by the client; the implementations of the objects, which are Web service clients, that the client will invoke; and the illusion capsules which will host these objects. On the other hand, the pattern WebServices-to-X will specify the rules to generate the Web services that will be the clients of the actual remote objects, wrapping their functionality.

In case of defining new grammar rules, we may define a new parser, which will be able to interpret the new defined platform specific patterns. This is also feasible to be done without the underlying implementation having to undergo any change. The reason is again our design choice to implement the particular parsers (*ArchXMLParser* and *PatternXMLParser* classes) as specializations of a basic *XMLParser* class. A new specialized parser can be incorporated as a XMLParser subclass, without affecting the existing parser implementations and functionality.

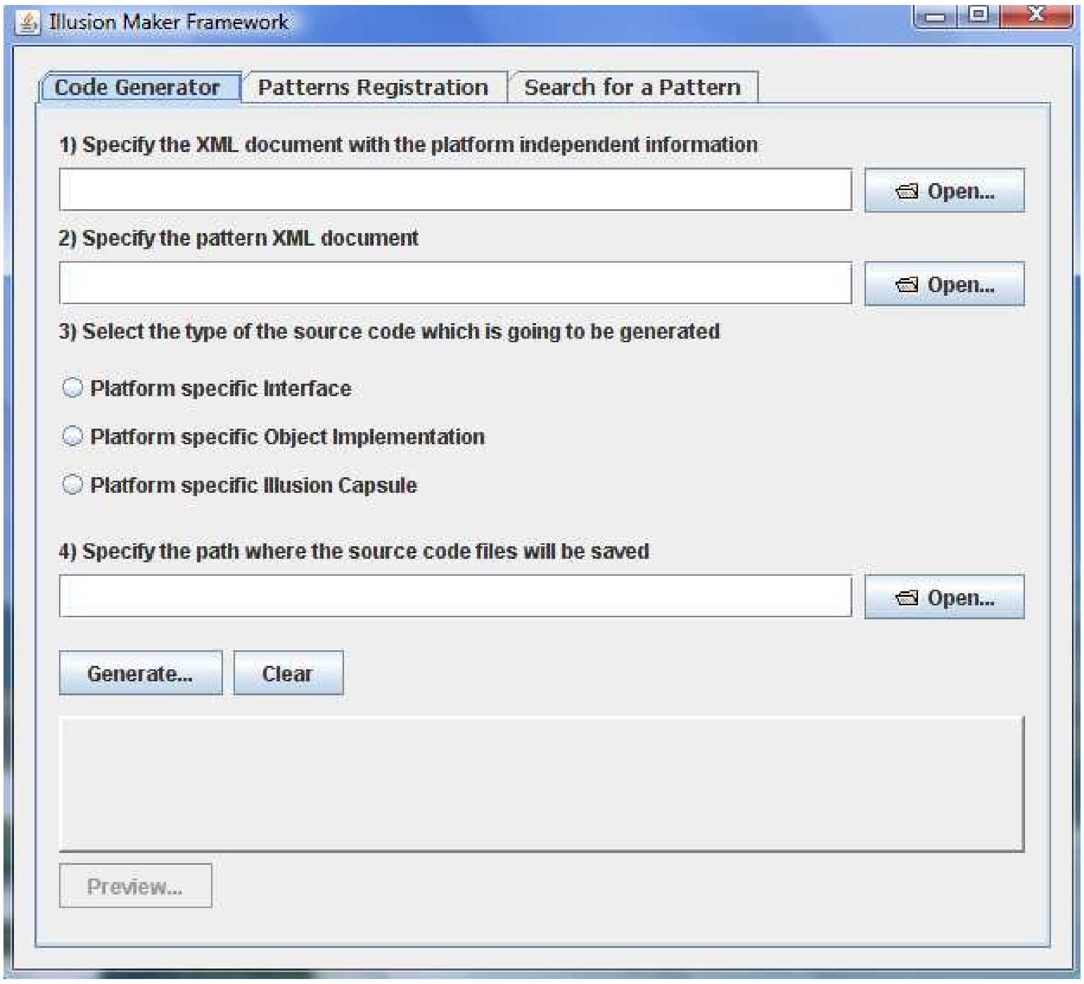
Moreover, we are able to implement new *ContentHandler* instances, which will offer the call back mechanism attached to the new parser. These instances will be specializations of the *DefaultHandler* adapter class, which they will override the *characters()*, *startElement()*, and *endElement()* methods. The parser will invoke back these methods, while receiving notification of the platform specific patterns content. The Illusion Maker framework is rendered scalable, by utilizing the interfaces offered by SAX API.

Finally, the new specified platform specific patterns will be ready to be used within the framework, to attain integration between the new and the existing middleware platforms. In our example scenario, supposing that we have a third server application on top of platform X then we will use the Corba-to-WebServices and WebServices-to-X patterns to generate the necessary source code. On the contrary, if a new client application, that has been implemented on the basis of platform X, joined the environment, we would use the X-to-WebServices and WebServices-to-RMI patterns.

3.7 The Graphical User Interface of the Framework

The Illusion Maker framework offers a graphical user interface (GUI) which incorporates the whole functionality. The user-friendly environment makes its use simple enough. Figure 3.11 shows the main window of the GUI which depicts the basic functionality of the code generation.

The user should specify in the first two fields the input components, that is the appropriate XML documents of the platform independent information and the platform specific patterns. In particular, the user searches and specifies the paths of the corresponding XML files.



1) Specify the XML document with the platform independent information

2) Specify the pattern XML document

3) Select the type of the source code which is going to be generated

- Platform specific Interface
- Platform specific Object Implementation
- Platform specific Illusion Capsule

4) Specify the path where the source code files will be saved

Figure 3.11: Graphical User Interface - Basic Functionality

The kind of the output source code must be specified by selecting one option each time in the radio boxes. These options correspond to the three possible types of source code which can be generated, as follows:

1. *Platform specific Interface*: This option corresponds to the generation of a platform specific illusion interface, which is supposed by the client objects.

2. *Platform specific Object Implementation*: This corresponds to the generation of the code which will act as the mediator to bring about the interoperability. In other words, this option should be selected for the generation of: either the platform specific objects on top of the middleware platform assumed by a client application (serving as Web service clients); or the Web services that will be the clients for the actual remote objects. The result depends on the type of the specified platform specific pattern.

3. *Platform specific Illusion Capsule*: This selection corresponds to the generation of a platform specific illusion capsule, which will host the objects invoked by the client application.

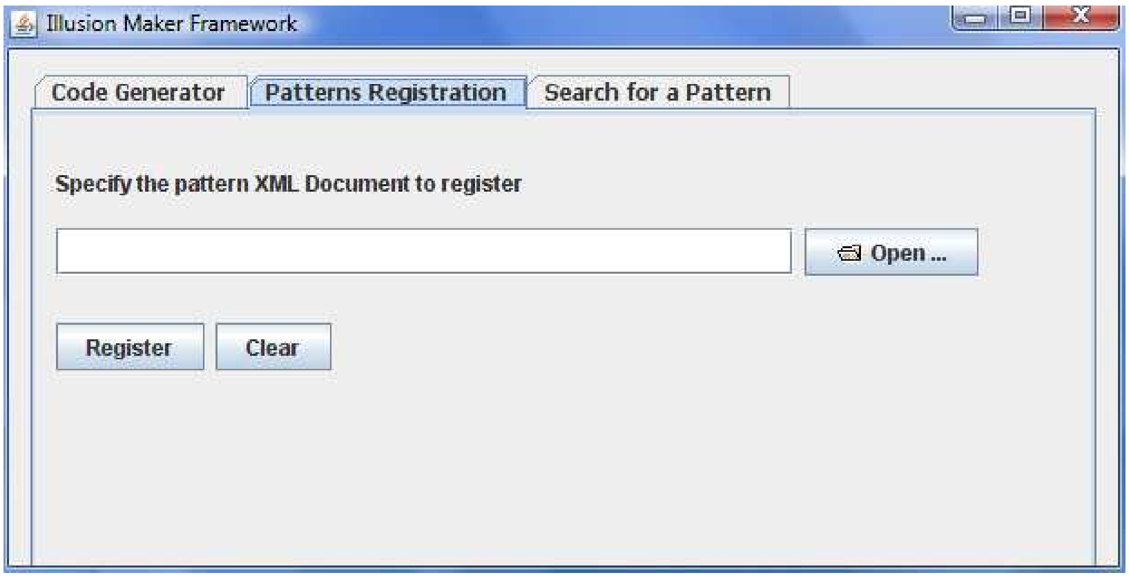


Figure 3.12: Graphical User Interface - Registration of Platform Specific Patterns

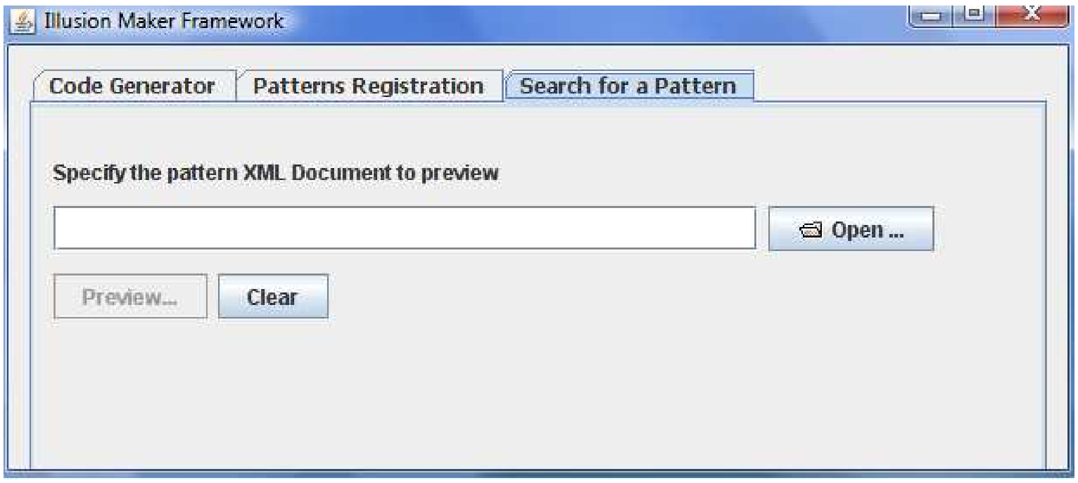


Figure 3.13: Graphical User Interface - Search for a Pattern

In any occasion, the XML document that contains the appropriate part of a platform specific pattern should be specified, in order to generate the corresponding source code. This means that, for creating a platform specific interface, the part of a *-to-WebServices pattern responsible for generating an interface declaration must be given as input (see listing 3.3). For the other cases, corresponding to each selection, listings 3.4, 3.6 and 3.5 can possibly be specified as input respectively.

Finally, the path within the file system, where the output files will be stored, can be selected and specified in the third text field. Provided that the source files have been successfully created, the environment gives the user the possibility to take a glance of the generated code by pressing *Preview* button, at the bottom of the window. A new window will be opened, containing the source code.

The graphical interface provides two other operations concerning the handling of the patterns, which supplement the main functionality of the framework. The first operation is related to the registration of platform specific patterns. The user can gather all the XML documents (either the ones of the platform independent information or those of the platform specific patterns) into a particular filesystem path, assumed by the framework. In this way, the patterns can easily be discovered in a particular place within the environment and then be used for the specific purpose. The second operation gives the possibility to search for a pattern XML document and preview its content. Figures 3.12 and 3.13 show the corresponding windows of the graphical interface, which provide these operations.

Chapter 4

Evaluation of the Illusion Maker Framework

To highlight the usefulness of the proposed framework, we rely on the experimental results coming from our case study scenario. In particular, our evaluation focuses on the following points:

1. The *implementation effort*, required for the development of the platform interoperable code.

2. The *overhead* which is introduced in the distributed system as a result of the utilization of the mediator.

The implementation effort reflects the gain from applying the proposed methodology, because the developers do not have to write the necessary source code manually. Instead, the necessary code is generated automatically by the code generator mechanism of the framework. The overhead depicts the delay which is introduced in the intercommunication between the heterogenous applications, while using the generated source code as a mediator. The implementation effort is measured in terms of the well-known *LOC* (Lines of Code) metric [17]. Specifically, we measure the amount of the necessary source code which is generated. The overhead is measured as the elapsed time, in *milliseconds*, from the moment the client calls a server operation to the moment the client receives back a response. We compare scenarios in which we make use of the Web service wrappers as a mediator with scenarios in which the client-side illusion objects access directly the actual remote objects, without using Web services. We also compare these cases with non-heterogenous cases. Finally, we point out the conclusions deduced from the experimental evaluation.

4.1 Implementation Effort

We formalize the *implementation effort* metric (LOC) as a function of parameters that reflect the scale of a server application involved in a heterogenous scenario. The size of these parameters is reflected by the metrics given in table 4.1, and the information about this size comes from the platform independent information. The implementation effort functions are used for measuring the total number of LOC generated by the framework. This measure depicts the impact in the implementation effort which a developer has to pay in case of implementing the mediator elements manually. The amount of generated LOC is compared with the size of the platform specific patterns. In this way, we highlight a *threshold*, which denotes the minimum size that the legacy server's parameters should have, so as to obtain a benefit from using the proposed framework.

Metric	Definition
N_{Interf}	Number of provided interfaces
N_{Caps}	Number of capsules
N_{Elem_i}	Number of elements hosted by the i-th capsule
N_{Op_i}	Number of operations provided by the i-th interface and implemented by the i-th element
$N_{Arg_{ij}}$	Number of parameters accepted by the j-th operation of the i-th element

Table 4.1: Metrics for the Size of the Server Application

In the case of our example scenario, the number of interfaces is $N_{Interf} = 2$, with one element realizing each of them; so for each interface, it is $N_{Elem} = 1$. Both the elements are hosted by the same capsule, that is $N_{Caps} = 1$. Moreover, one operation is provided by every one interface and is implemented by a corresponding element; so $N_{Op} = 1$ per interface and per element. Each of these operations accepts one parameter, hence $N_{Arg} = 1$ per operation.

4.1.1 Formalization

We use the CORBA-to-WebServices pattern (listings 3.3, 3.4 and 3.5) and the WebServices-to-RMI pattern (listing 3.6) to derive the implementation effort functions, which can be applied in a case like the one mentioned in our example scenario, for calculating the generated amount of LOC. In the same manner, we can derive further implementation effort functions for other cases of platform specific patterns (see appendix).

Considering the number of CORBA-specific illusion interfaces that must be generated, we have $(N_{Op} + 2)$ LOC per interface. Hence, the LOC for the total number of interfaces is:

$$LOC_{Interf} = 2N_{Interf} + \sum_{i=1}^{N_{Interf}} N_{Op_i} \tag{4.1}$$

Regarding the implementations of CORBA-specific elements which realize the illusion interfaces, for each element we have a standard number of *23 LOC*. In addition, for each operation which is implemented by an element, we have $(N_{arg} + 17)$ LOC in case of a non-void operation, and $(N_{arg} + 15)$ LOC, in case of a void operation. Taking the overall number of operations into account, we have $[(17N_{Op}^{(nonvoid)} + \sum_{j=1}^{N_{Op}^{(nonvoid)}} N_{Arg_j}) + (15N_{Op}^{(void)} + \sum_{k=1}^{N_{Op}^{(void)}} N_{Arg_k})]$ LOC generated. Therefore, the LOC for all the implementations of elements is:

$$LOC_{Elem} = 23N_{Elem} + 17\sum_{i=1}^{N_{Elem}} N_{Op_i}^{(nonvoid)} + 15\sum_{i=1}^{N_{Elem}} N_{Op_i}^{(void)} + \sum_{i=1}^{N_{Elem}}\sum_{j=1}^{N_{Op_i}} N_{Arg_{ij}} \tag{4.2}$$

As far as the code for the CORBA-specific illusion capsules is concerned, we have *29 LOC* for each hosting element; hence for the overall number of elements hosted by a capsule, $29N_{Elem}$ *LOC* are generated in total. Consequently, for all the illusion capsules required in a distributed environment, the total number of LOC is:

$$LOC_{Capsule} = 29\sum_{i=1}^{N_{Caps}} N_{Elem_i} \tag{4.3}$$

Concerning the Web service wrappers, that have to be generated by utilizing the WebServices-to-RMI pattern, for each Web service we have a standard number of *5 LOC*. Furthermore, for each non-void and void wrapping operation we have *16* and *14 LOC* respectively. So, every Web service consists of $(5 + 16N_{Op}{}^{(nonvoid)} + 14N_{Op}{}^{(void)})$ LOC totally. Since for every element a corresponding Web Service wrapper is generated, then the overall number of LOC which is generated is:

$$LOC_{WS} = 5N_{Elem} + 16\sum_{i=1}^{N_{Elem}} N_{Op_i}^{(nonvoid)} + 14\sum_{i=1}^{N_{Elem}} N_{Op_i}^{(void)} \tag{4.4}$$

So, the total *LOC* which is generated by the framework, using this couple of platform specific patterns, is the aggregation of the amount coming from the four aforementioned equations:

$$LOC_{Total} = LOC_{Interf} + LOC_{Elem} + LOC_{Capsule} + LOC_{WS} \tag{4.5}$$

In the case of our example scenario, by applying the aforementioned implementation effort formula, we result in the amount of generated LOC given in table 4.2. This number does not make any sense, compared to the size of platform specific patterns. However, this is just a simple case study scenario, involved in a small scale distributed environment. In the following subsection, we show that the benefit arising from the application of the Illusion Maker framework is significant, as the scale of a heterogenous distributed system rises.

LOC_{Interf}	6
LOC_{Elem}	78
$LOC_{Capsule}$	58
LOC_{WS}	38
$\mathbf{LOC_{Total}}$	**180**

Table 4.2: Implementation Effort for our Case Study

4.1.2 Experimental Results

We investigate the implementation effort gained by applying the proposed approach with respect to the size of the platform specific patterns that should be developed. Figure 4.1 shows the amount of generated LOC for a CORBA interface declaration (equation 4.1), as the number of operations increases. The size of the part of the CORBA-to-WebServices pattern, responsible for the generation of these interface declarations, is equal to *34 LOC* (see listing 3.3). Supposing a case where only one interface is provided by the actual server application, we observe that a developer obtains a gain if the actual interface provides 33 operations at least. This seems negative at a first glance. However, it does not reflect the whole case.

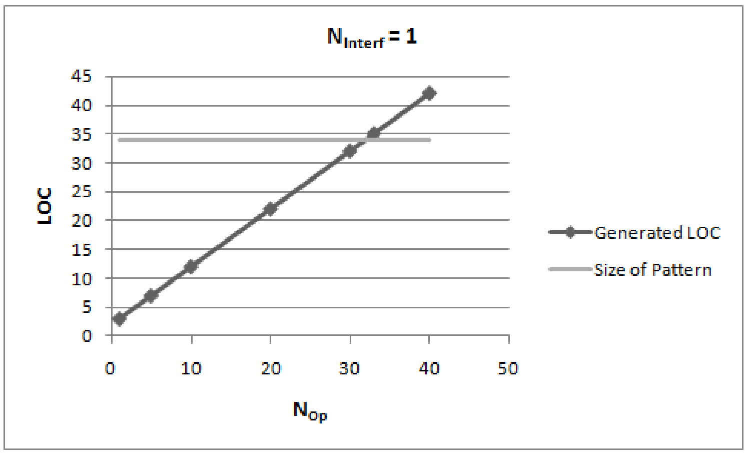


Figure 4.1: Implementation Effort for the Generation of CORBA Illusion Interfaces

The next two charts highlight the generated amount of LOC for a CORBA object implementation (equation 4.2). The first chart (figure 4.2) shows the amount of LOC as a function of the number of elements, realizing a CORBA illusion interface. Supposing that the actual interface provides one non-void operation, which accepts one parameter, we observe that if 3 elements exist at least, realizing the actual server's interface, the use of our framework for the code generation benefits a developer. The size of the corresponding part of the CORBA-to-WebServices pattern (see listing 3.4) is equal to *117 XML LOC*.

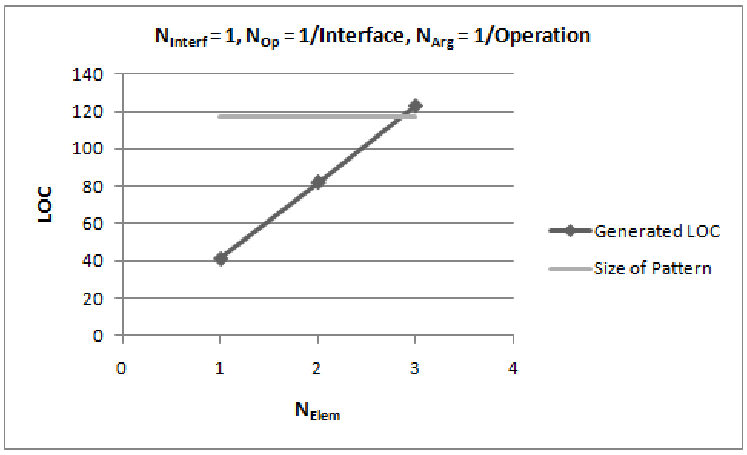


Figure 4.2: Implementation Effort for the Generation of CORBA Object Implementations - 1

The second chart (figure 4.3) shows the amount of LOC as a function of the number of operations. We suppose that one CORBA element realizes a CORBA illusion interface, and each provided operation accepts one parameter. We assume non-void operations only. In this case, we see that the actual server element must implement 6 operations at least, so as to obtain a gain with respect to the amount of generated code.

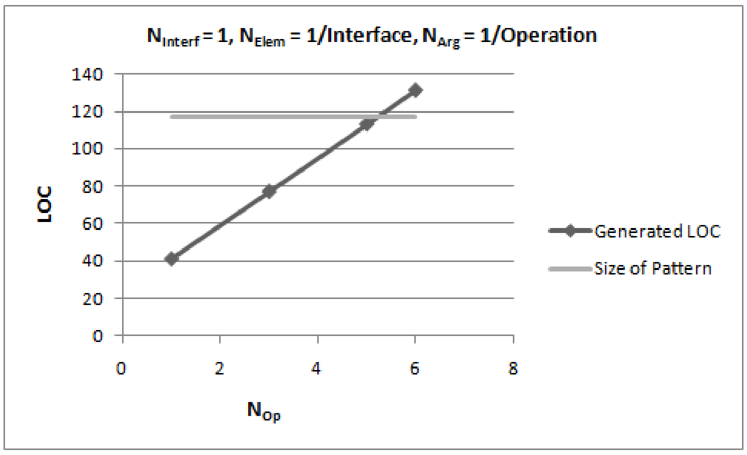


Figure 4.3: Implementation Effort for the Generation of CORBA Object Implementations - 2

Following that, we give the representation of the amount of LOC needed for a CORBA-specific illusion capsule (equation 4.3), as a function of the number of CORBA elements that the capsule hosts (figure 4.4). The responsible for the generation of these illusion capsules part of the CORBA-to-WebServices pattern has a size of *62 XML LOC* (see listing 3.5) . We perceive that as the number of hosting elements rises, the amount of LOC rises proportionately. A developer gains in the existence of 3 hosting elements by the actual server capsule.

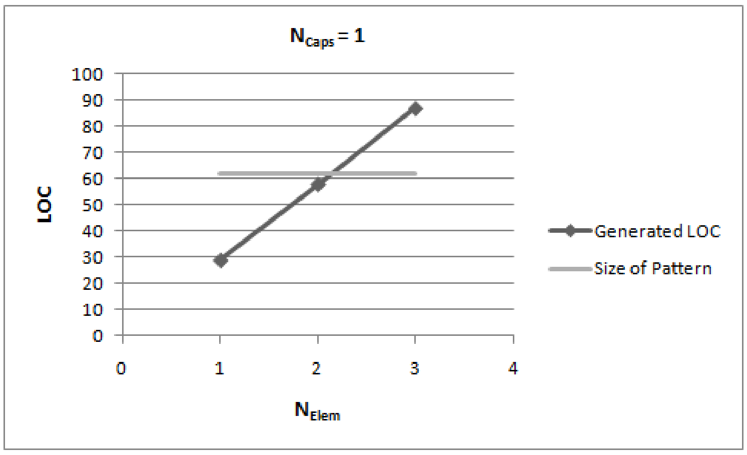


Figure 4.4: Implementation Effort for the Generation of CORBA Illusion Capsules

Regarding the Web service wrappers acting as Java RMI clients, that have to be generated for every element of the actual server application, the corresponding WebServices-to-RMI platform specific pattern consists of *85 XML LOC* (see listing 3.6). Figure 4.5 gives the generated amount of LOC as a function of the number of elements, while figure 4.6 gives the amount of LOC as a function of the number of operations, implemented by a server element (equation 4.4). We assume again non-void operations.

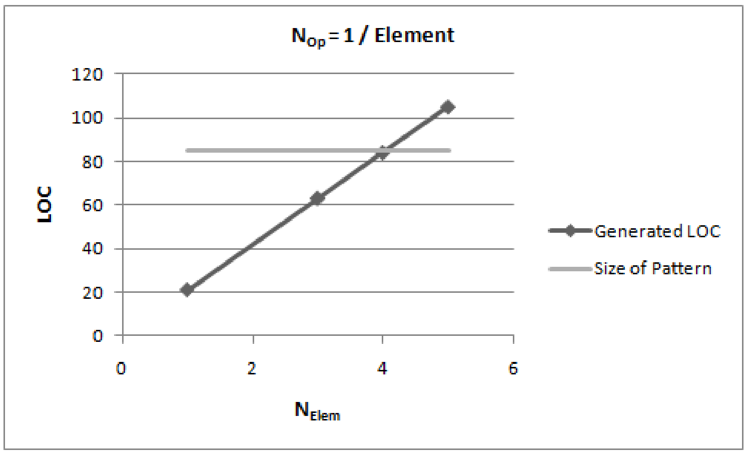


Figure 4.5: Implementation Effort for the Generation of Web Service Wrappers - 1

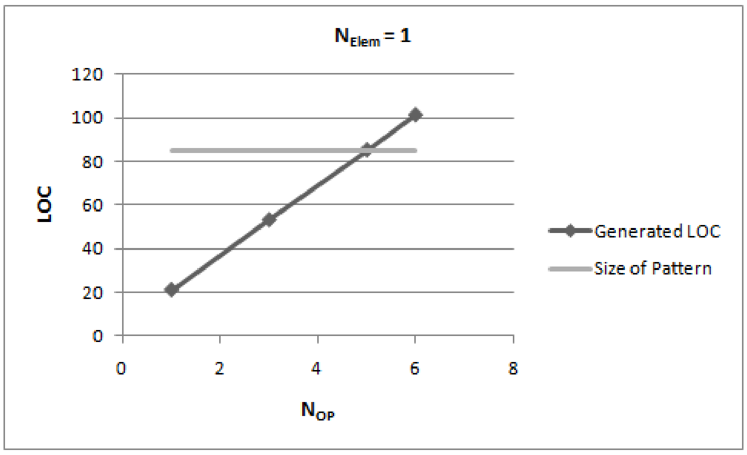


Figure 4.6: Implementation Effort for the Generation of Web Service Wrappers - 2

In the first case, we consider that each actual server element implements one operation. If the actual server application consists of 5 elements at least, we reap the benefit from using the Illusion Maker for the generation of the Web service wrappers. In the second case, we suppose that the server application has one element. As the number of operations implemented by this element increases, we observe that 6 operations are adequate for a developer to make a gain with respect to the implementation effort.

Finally, the chart in figure 4.7 provides a total view concerning the implementation effort. We assume a case where an actual Java RMI server provides one remote interface containing non-void operations that accept one parameter. One Java RMI-specific element realizes the aforementioned interface and one server capsule hosts the element. We show the total amount of LOC, which is necessary to be generated for the previous scenario, as a function of the number of provided operations. The overall size of the platform specific patterns to be used is equal to *298 XML LOC*. We perceive that the server application should provide at least 7 operations. In such a case, the developer obtains the benefit from utilizing our framework to achieve the integration.

From the previous results, we further observe that the implementation of illusion elements is the most demanding for someone to develop, with the implementation of Web service wrappers following after. These two cases are expected to be the most demanding as they require the development of the operations that serve as Web service clients and as clients for the actual remote objects, respectively. On the other hand, the implementation effort for the illusion interface declarations is the less demanding for a developer to pay. We can also point out that the factors that contribute mostly to the increase of LOC are the N_{Elem} and the N_{Op}. Consequently, as the scale of a heterogenous distributed system increases, the utilization of the Illusion Maker framework gives us a substantial advantage.

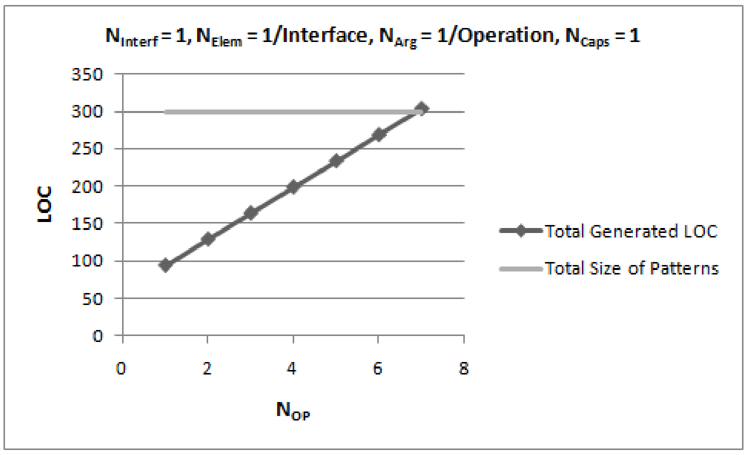


Figure 4.7: Total Implementation Effort for achieving Integration

4.2 Overhead

The use of a mediator for the integration of the heterogenous distributed applications introduces inevitably a delay in the communication between the client and the server objects. This delay degrades the performance of the whole system and may be substantial, depending on the kind of the services provided and on the needs of the clients. We evaluate the overhead by measuring the total time elapsed until the client receives back the response. We also try to portion out the total overhead among the components involved in the intercommunication, and find out which of them contribute the most. Our evaluation approach focuses mainly on the invocation cost. The components, acting as clients, initialize once the connections and execute a recurrent call to the corresponding server components. Then, we find an average value of the response time involved in each case. This approach provides a more representative view, concerning the factors that contribute most to the overhead. The experiments were performed in an environment where the actual client and server element resided in different machines, connected through a Local Area Network Ethernet switch of 100 Mbps speed.

4.2.1 Non-Heterogenous Distributed Environments

We start with the invocation costs involved in cases where both the client and the server elements rely on the same middleware. Table 4.3 gives the results. For the CORBA and Java RMI cases the client and server elements reside in different hosts. The clients make a static call in both cases. For the Web service case, the client constructs at run time a call to a locally deployed Web service. We observe that the invocation cost is relatively low for Java RMI, more higher for CORBA and quite more higher for the dynamic Web service invocation.

CORBA Response Time = 7.937 ms/call
RMI Response Time = 2.6525 ms/call
Web Service Response Time = 14.9235 ms/call

Table 4.3: Average Response Times in Non-Heterogenous Environments

4.2.2 First Scenario: A CORBA Client invokes a Java RMI Server

The scenario of figure 4.8 corresponds to the one depicted by our case study. The two elements hosted by the client and the server capsules are the legacy applications. We deploy both the CORBA-specific illusion object, acting as Web service client, and the Web service wrapper on the server machine.

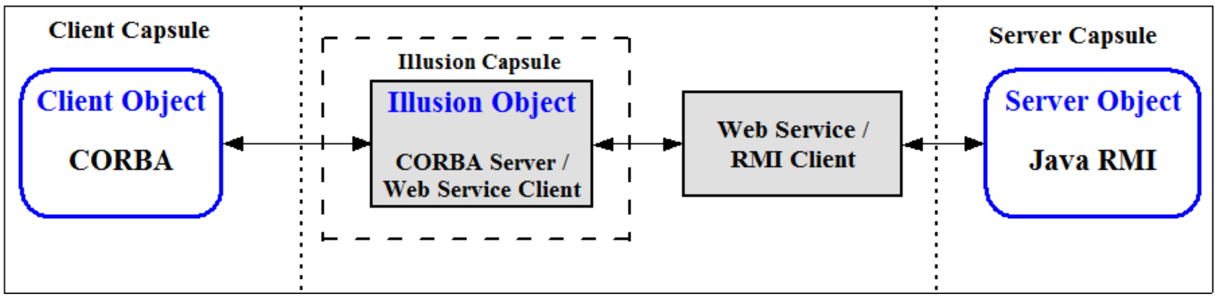


Figure 4.8: First Scenario: Using Web Services

Table 4.4 gives the experimental results. The **RMI Response** reflects the time elapsed until the Web service, acting as RMI client, receives back the response from the actual RMI server object. The **Web Service Response** reflects the time elapsed until the CORBA-specific illusion server object takes the response back from the Web service. It is obvious that this time interval includes the RMI response time period. The **CORBA Response** represents the total delay between the moment the actual client object makes the request and the moment it receives the response. From these results, we deduce the following:

- Each RMI request-response takes 0.517 ms;
- each Web service request-response takes 7.117 - 0.517 = 6.6 ms;
- each CORBA request-response takes 15.594 - 7.117 = 8.477 ms.

RMI Response Time = 0.517 ms/call
Web Service Response Time = 7.117 ms/call
CORBA Response Time = 15.594 ms/call

Table 4.4: First Scenario: Average Response Times using Web Services

Next we compare these results with the ones coming from the case in which we do not use Web services as mediator. Instead, the CORBA-specific illusion object acts as a RMI client for the actual server object. Figure 4.9 depicts the distributed environment for this case, with the illusion object being deployed on the client machine. In this way, the client object makes a local call to the illusion object, which is the proxy that forwards the request to the server object.

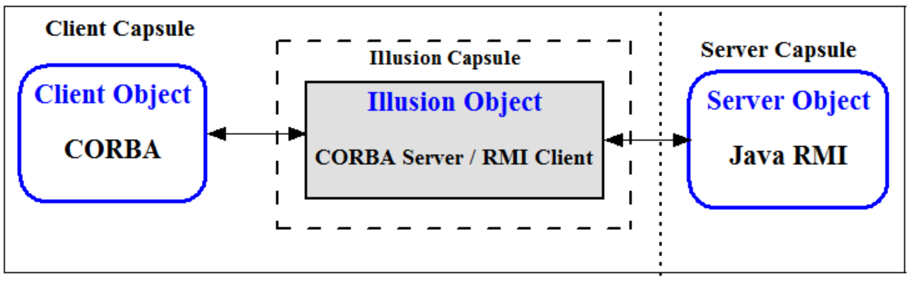


Figure 4.9: First Scenario: The CORBA-specific Illusion Object access directly the Server Object

Table 4.5 gives the results for the average response times, without using Web services. In this case we perceive that:

- The RMI response takes more time than in the first case (1.15 vs 0.517 ms), while the CORBA response time is smaller than the first case: 8.556 - 1.15 = 7.406 vs 8.477 ms.
- Regarding the total execution time, the use of Web services introduces an important overhead in the total delay of the side-to-side communication. This overhead is about 45% higher (15.594 vs 8.556 ms) in the first case compared to the second case.

RMI Response Time = 1.15 ms/call
CORBA Response Time = 8.556 ms/call

Table 4.5: First Scenario: Average Response Times without using Web Services

4.2.3 Second Scenario: A Java RMI Client invokes a CORBA Server

Figure 4.10 gives a scenario where the actual client is based on Java RMI and the actual server is based on CORBA. We use a Web service wrapper for the integration. Both the RMI-specific illusion object and the Web service wrapper are deployed on the server capsule. Table 4.6 gives the experimental results for the execution times of operation invocations. In this case, we have larger response times than the corresponding case of the opposite scenario, in the previous subsection:

- Each CORBA request-response takes 2.9962 ms;
- each Web service request-response takes 65.1452 - 2.9962 = 62.1490 ms;
- each RMI request-response takes 90.6320 - 65.1452 = 25.4868 ms.

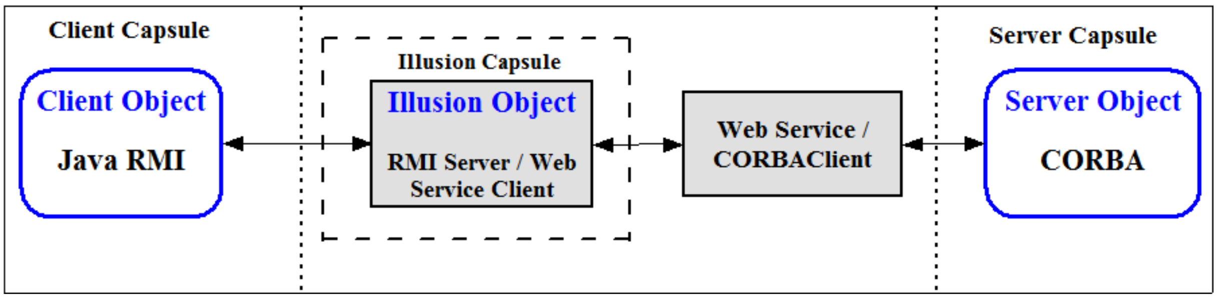


Figure 4.10: Second Scenario: Using Web Services

CORBA Response Time = 2.9962 ms/call
Web Service Response Time = 65.1452 ms/call
RMI Response Time = 90.6320 ms/call

Table 4.6: Second Scenario: Average Response Times using Web Services

Table 4.7 gives the corresponding results for the same scenario, but without using Web service. The RMI-specific illusion object invoked by the actual client acts as a CORBA client accessing the actual server. The illusion object is deployed on client capsule (figure 4.11). Again, the execution time of the invocation is substantially larger compared to the correspinding case of the opposite scenario:

- CORBA response takes 3.7854 ms, while RMI response takes 64.1797 - 3.7854 = 60.3943 ms.
- Concerning the total execution time for the second scenario, the overhead is about 30% higher in the case of using Web service (90.6320 vs 64.1797 ms).

CORBA Response Time = 3.7854 ms/call
RMI Response Time = 64.1797 ms/call

Table 4.7: Second Scenario: Average Response Times without using Web Services

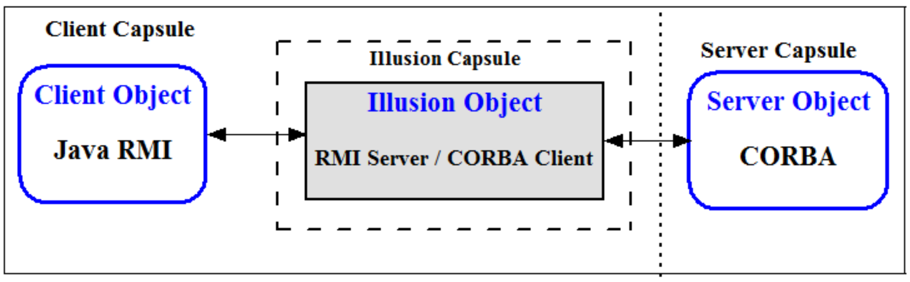


Figure 4.11: Second Scenario: The RMI-specific Illusion Object access directly the Server Object

4.2.4 Comparison

The comparison focuses on the following points:

- The introduced overhead in a heterogenous distributed environment compared with the typical overhead existing in a non-heterogenous environment.
- The overhead in the case of using Web service wrappers versus the overhead in the case of not using Web service wrappers, as the mediator which realizes the interoperability.

The first chart (figure 4.12) shows the expected difference in the overhead between the heterogenous and non-heterogenous cases. In a non-heterogenous environment the overhead arises mainly from the network. On the other hand, in a heterogenous environment the interoperable software elements introduce extra overhead. The second chart (figure 4.13) shows the substantially larger overhead that is caused by the use of Web services, as the mediator element.

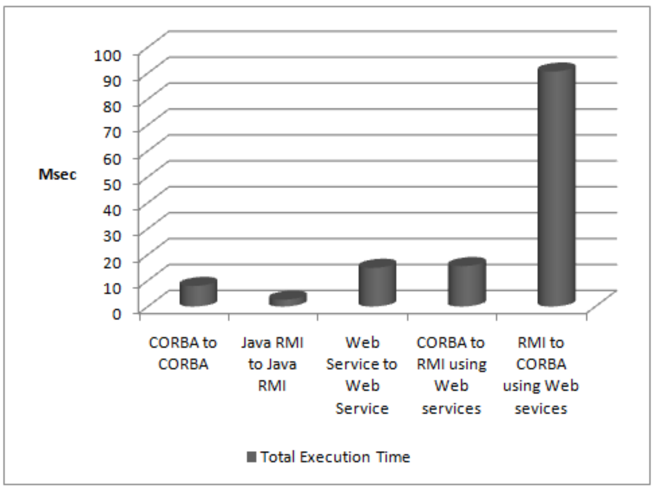


Figure 4.12: Heterogenous vs Non-Heterogenous Distributed Environment

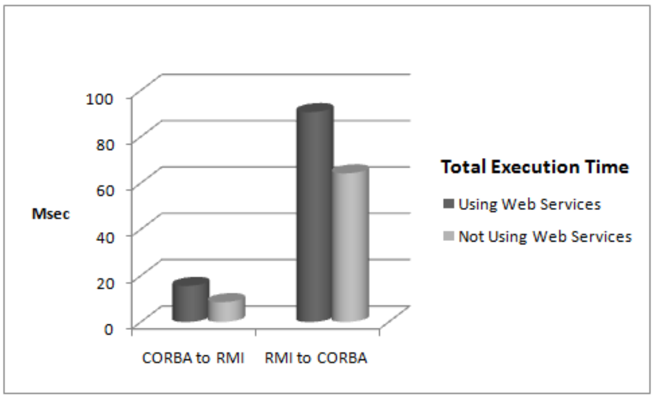


Figure 4.13: Using Web Services vs Not Using Web Services

4.3 Conclusion

In the developer's perspective, the benefit from using the Illusion Maker framework is unquestionable. Someone who wishes to implement the components that will make the integration of heterogenous applications possible saves significant time, specifically in the case of large-scale distributed systems. He/she is released from the obligation to have always to write manually the customary source code, responsible for the interoperability. On the contrary, he/she has the chance to devote more time about issues concerning the implementation of the functionality of the distributed applications. The code generator mechanism produces quickly the necessary code for the integration. Afterwards, the developer has this code at his/her disposal, being free to implement the desirable functionality. Following that, one can compile the code, produce the corresponding executable files and deploy them on the suitable hosts. The user is hereafter able to utilize the integrated distributed applications. He/she can use heterogenous services offered by the environment through his/her own application, in the same manner as if the assuming middleware infrastructures were the same.

Regarding the overhead, we have to take into consideration not only the delay introduced by the utilization of the mediator, but also the network overhead (e.g. the workload, the throughput, the bandwidth). Depending on the purpose of the provided functionality, the capability of the applications to serve quickly the clients' requests may be crucial. So, we have to examine solutions in order to minimize the extra overhead.

In the previous section, we saw that the use of the Web service wrappers introduces a significant overhead. On the other hand, in section 3.6 we emphasized that the use of the Web services provides the framework with an important degree of scalability, as regards design issues for future extensions of its functionality. Hence, we have a trade-off concerning the use of Web services, and for this reason we should try to minimize the negative effect of the overhead. We might pay more attention on the issue of the deployment of the components that bring the interoperability. Specifically, we can examine where the interoperable elements are better to be deployed, i.e. on the client capsule or on the server capsule. If the functionality of a distributed application is critical with respect to the execution time involved, the use of Web services as the mediator might be avoided. In this case, it would be better to prefer a solution like the ones depicted by figures 4.9 or 4.11.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we discussed the problem of interoperability between distributed applications that have been implemented on top of heterogenous middleware infrastructures. We proposed the Illusion Maker framework that aims at enabling middleware platform interoperability, without imposing any particular constraint on the middleware platforms, used for the development of the distributed applications. Given a client application that relies on a particular middleware platform and a server application that relies on another middleware platform, the Illusion Maker creates the "illusion" that the server application relies on the platform assumed by the client application.

The framework automates the process of generating the source code of the software elements that constitute the aforementioned illusion. The code generator mechanism does not depend on the platforms assumed by the client or the server. Specifically, the generation of illusions relies on platform specific patterns which are given as input to the Illusion Maker framework. These patterns specify a set of rules that model the descriptions of mappings between the different middleware standards. In this way, the proposed approach is rendered generic enough to be used in any distributed environment.

We chose to use XML for the specification of the platform specific patterns. In this way, the parsing of the patterns was facilitated. We implemented a SAX parser, which accepts as input an XML-based description of a platform specific pattern and converts it into a list of objects, belonging to a particular class hierarchy. To define this class hierarchy, we applied the Interpreter design pattern. Also, the Interpreter design pattern simplified the implementation of the code generation process.

The evaluation of the proposed methodology focused on the benefit for a developer, following from the fact that he/she does not have to write the necessary source code manually. The framework produces the interoperable source code quickly. A developer has at his disposal this code, saving significant time which he/she can devote to implement the desirable functionality. In addition, we examined the issue of the overhead, which is introduced inevitably by adding the interoperable software elements to a distributed environment. The experimental results showed that the use of Web services as the intermediate element for the integration causes an important increase in the time needed for the execution of a client request.

5.2 Future Work

From an engineering point of view, one can further explore the following issues:

- The extension of the framework in order to be able to handle complex data types and exceptions.
- The capability of the already defined pattern rules to specify new patterns in case of using other middleware platforms (in addition to CORBA and Java RMI).
- The interoperability issues introduced by the use of different middleware services (e.g. the CORBA Object Transaction Service) on the side of the client that differ from the middleware services assumed on the side of the server (e.g. the Java Transaction Service). It would be a challenge, indeed, to define patterns for this kind of services.
- The possibility to replace the XML descriptions with other more user-friendly descriptions.

Regarding the performance of the distributed applications, one could examine the issue of deployment of the components that bring the interoperability. If these components are deployed on client-side, the performance may be enhanced. However, this deployment causes security issues concerning the client machine. On the other hand if the components are deployed on an intermediate machine or on the server machine, the performance may degrade. Finally, the possibility of using other middleware platforms in place of Web services with respect to the issue of performance, could be investigated.

Bibliography

[1] E. Aarts, R. Harwig and M. Schuurmans, *Ambient Intelligence, chapter The Invisible Future: The Seamless Integration of Technology into Everyday Life*, pp. 235-250. McGraw-Hill, 2001.

[2] M. Aleksy, A. Korthaus, M. Schader, *Implementing Distributed Systems with Java and Corba*, Springer, June 2005, available at http://www.springerlink.com/content/n4v226/?p=8b685689f0494cf1ae95040f1be65256&pi=0

[3] *The Apache Software Foundation*, WebServices-Axis, Java Axis API Documentation for Apache Axis 1.2, May 2005, http://ws.apache.org/axis/java/apiDocs/index.html

[4] P. A. Bernstein, Middleware: A Model for Distributed System Services, *Communications of the ACM*, 39(2), pp. 86-98, Feb. 1996.

[5] Y.-D. Bromberg and V. Issarny, INDISS: Interoperable Discovery System for Networked Services, *In Proceedings of Middleware 2005*, pp. 164-183, 2005.

[6] W. Emmerich, *Engineering Distributed Objects*, John Wiley & Sons, Ltd, 2000.

[7] P. Falcarin and G. Alonso, Software Architecture Evolution through Dynamic AOP. In F. Oquendo, B. Warboys, and R. Morrison, editors, *Software Architecture: First European Workshop (EWSA 2004)*, volume 3047 of LNCS, pp. 57-73. Springer, 2004.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Longman, 1995.

[9] P. Grace, G. S. Blair and S. Samuel, A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments, *ACM Mobile Computing and Communications Review*, 9(1), pp. 2-14, 2005.

[10] E. R. Harold, *Processing XML with Java*, 2002, Online Edition available at http://www.cafeconleche.org/books/xmljava/.

[11] ISO/IEC. *Open Distributed Processing Reference Model, Part3: Architecture.* Technical Report 10746-3, ISO/IEC, 1995.

[12] M. Morrison, *Teach Yourself XML in 24 Hours*, Second Edition, by Sams Publishing, 2002.

[13] Object Management Group Available Specification, *IDL to Java Language Mapping*, Version 1.3, OMG Document Number: formal/08-01-11, January 2008, http://www.omg.org/cgi-bin/doc?formal/08-01-12

[14] Object Management Group Available Specification, *Java to IDL Language Mapping*, Version 1.4, OMG Document Number:formal/08-01-14, January 2008, http://www.omg.org/cgi-bin/doc?formal/08-01-14

[15] Sun Microsystems Inc., JSR-101 Expert Group, *Java(TM) API for XML-based RPC JAX-RPC Specification*, Version 1.1, October 2003, http://jcp.org/aboutJava/communityprocess/final/jsr101/index2.html

[16] A. Zarras, A Comparison Framework for Middleware Infrastructures, *Journal of Object Technology*, 3(5), pp. 100-123, 2004.

[17] A. Zarras, Applying Model Driven Architecture to Achieve Distribution Transparencies, *Information and Software Technology*, 48(7), pp. 498-516, 2006.

Appendix

RMI-to-WebServices Pattern

Listing 5.1: Generation of Java RMI Illusion Interfaces

```
<?xml version="1.0"?>
<pattern type="Interf" platform="RMI">
  <datatypes>
    <datatype id="0" name="void">void</datatype>
    <datatype id="1" name="boolean">boolean</datatype>
            <datatype id="2" name="char">char</datatype>
            <datatype id="3" name="string">String</datatype>
            <datatype id="4" name="byte">byte</datatype>
            <datatype id="5" name="unsigned byte">short</datatype>
            <datatype id="6" name="short">short</datatype>
            <datatype id="7" name="unsigned short">short</datatype>
            <datatype id="8" name="int">int</datatype>
            <datatype id="9" name="unsigned int">int</datatype>
            <datatype id="10" name="long">long</datatype>
            <datatype id="11" name="unsigned long">long</datatype>
            <datatype id="12" name="float">float</datatype>
            <datatype id="13" name="double">double</datatype>
            <datatype id="14" name="BigInt">java.math.BigInteger</
                datatype>
            <datatype id="15" name="decimal">java.math.BigDecimal</
                datatype>
  </datatypes>
  <interf_exp>
    <literal_exp>import java.rmi.*;</literal_exp>
    <literal_exp>public interface </literal_exp>
    <interf_name/>
    <literal_exp> extends Remote {</literal_exp>
    <operation_exp>
      <retType/>
      <opName/>
      <literal_exp>(</literal_exp>
      <arg_exp>
        <direction_type_argName/>
        <literal_exp>,</literal_exp>
      </arg_exp>
      <literal_exp>) throws RemoteException;</literal_exp>
```

```
        </operation_exp>
        <literal_exp>_newline</literal_exp>
        <literal_exp>}</literal_exp>
        <literal_exp>_newline</literal_exp>
    </interf_exp>
</pattern>
```

Listing 5.2: Generation of Java RMI-specific Object Implementations

```
<?xml version="1.0"?>
<pattern type="ObjectImpl" platform="RMI">
  <datatypes>
    <datatype id="0" name="void" retValue="null" xmltype="AXIS_VOID"
        javalangtype="null">void</datatype>
                <datatype id="1" name="boolean" retValue="false" xmltype="
                    XSD_BOOLEAN" javalangtype="Boolean">boolean</datatype>
                <datatype id="2" name="char" retValue="'\u0000'" xmltype="
                    XSD_STRING" javalangtype="Character">char</datatype>
                <datatype id="3" name="string" retValue="null" xmltype="
                    XSD_STRING" javalangtype="String">String</datatype>
                <datatype id="4" name="byte" retValue="-1" xmltype="
                    XSD_BYTE" javalangtype="Byte">byte</datatype>
                <datatype id="5" name="unsigned byte" retValue="-1" xmltype
                    ="XSD_UNSIGNEDBYTE" javalangtype="Short">short</datatype
                    >
                <datatype id="6" name="short" retValue="-1" xmltype="
                    XSD_SHORT" javalangtype="Short">short</datatype>
                <datatype id="7" name="unsigned short" retValue="-1"
                    xmltype="XSD_UNSIGNEDSHORT" javalangtype="Short">short</
                    datatype>
                <datatype id="8" name="int" retValue="-1" xmltype="XSD_INT"
                     javalangtype="Integer">int</datatype>
                <datatype id="9" name="unsigned int" retValue="-1" xmltype=
                    "XSD_UNSIGNEDINT" javalangtype="Integer">int</datatype>
                <datatype id="10" name="long" retValue="-1" xmltype="
                    XSD_LONG" javalangtype="Long">long</datatype>
                <datatype id="11" name="unsigned long" retValue="-1"
                    xmltype="XSD_UNSIGNEDLONG" javalangtype="Long">long</
                    datatype>
                <datatype id="12" name="float" retValue="-1.0" xmltype="
                    XSD_FLOAT" javalangtype="Float">float</datatype>
                <datatype id="13" name="double" retValue="-1.0" xmltype="
                    XSD_DOUBLE" javalangtype="Double">double</datatype>
                <datatype id="14" name="BigInt" retValue="java.math.
                    BigInteger.ZERO" xmltype="XSD_INTEGER" javalangtype="
                    java.math.BigInteger">java.math.BigInteger</datatype>
                <datatype id="15" name="decimal" retValue="new java.math.
                    BigDecimal(java.math.BigInteger.ZERO)" xmltype="
                    XSD_DECIMAL" javalangtype="java.math.BigDecimal">java.
                    math.BigDecimal</datatype>
        </datatypes>
  <parametermodes>
    <mode name="in">IN</mode>
                <mode name="out">OUT</mode>
                <mode name="inout">INOUT</mode>
        </parametermodes>
  <element_exp>
    <literal_exp>import java.rmi.*;</literal_exp>
    <literal_exp>import java.rmi.server.*;</literal_exp>
    <literal_exp>import org.apache.axis.client.Call;</literal_exp>
    <literal_exp>import org.apache.axis.client.Service;</literal_exp>
    <literal_exp>import org.apache.axis.encoding.XMLType;</literal_exp>
    <literal_exp>import org.apache.axis.utils.Options;</literal_exp>
    <literal_exp>import javax.xml.rpc.ParameterMode;</literal_exp>
    <literal_exp>import java.util.Properties;</literal_exp>
    <literal_exp>import java.net.*;</literal_exp>
    <literal_exp>import java.lang.*;</literal_exp>
    <literal_exp>public class </literal_exp>
    <element_name/>
    <literal_exp> extends UnicastRemoteObject implements </literal_exp>
    <interf_name/>
    <literal_exp> {</literal_exp>
    <literal_exp>private String endpoint;</literal_exp>
    <literal_exp>public </literal_exp>
    <element_name/>
    <literal_exp>(String servURL) throws RemoteException {</literal_exp>
    <literal_exp>endpoint = servURL;</literal_exp>
    <literal_exp>}</literal_exp>
    <operation_exp>
      <literal_exp>public </literal_exp>
      <retType/>
      <opName/>
      <literal_exp>(</literal_exp>
      <arg_exp>
        <direction_type_argName/>
        <literal_exp>,</literal_exp>
      </arg_exp>
      <literal_exp>) {</literal_exp>
      <literal_exp>try {</literal_exp>
      <literal_exp>Service service = new Service();</literal_exp>
      <literal_exp>Call call = (Call) service.createCall();</literal_exp>
      <literal_exp>call.setTargetEndpointAddress(new URL(endpoint));</
          literal_exp>
      <literal_exp>call.setOperationName("</literal_exp>
      <opName/>
      <literal_exp>");</literal_exp>
      <addparameter_exp>
        <literal_exp>call.addParameter("arg</literal_exp>
        <arg_id/>
        <literal_exp>", XMLType.</literal_exp>
        <type/>
        <literal_exp>, ParameterMode.</literal_exp>
        <direction/>
        <literal_exp>);</literal_exp>
      </addparameter_exp>
      <literal_exp>call.setReturnType(XMLType.</literal_exp>
      <retType/>
      <literal_exp>);</literal_exp>
```

```
        <invoke>
          <retType_exp id="1">
            <retType/>
            <literal_exp> ret = (</literal_exp>
            <retType/>
            <literal_exp>) </literal_exp>
          </retType_exp>
          <literal_exp>call.invoke(new java.lang.Object[] { </literal_exp>
          <arg_exp>
            <direction_type_argName/>
            <literal_exp>,</literal_exp>
          </arg_exp>
          <retType_exp id="2">
            <literal_exp>_tab</literal_exp>
            <literal_exp>return ret;</literal_exp>
          </retType_exp>
        </invoke>
        <literal_exp>} </literal_exp>
        <literal_exp>catch(Exception ex) {</literal_exp>
        <literal_exp>System.err.println("ERROR: " + ex);</literal_exp>
        <literal_exp>ex.printStackTrace(System.out);</literal_exp>
        <literal_exp>System.out.println("Cannot establish connection with Web
            Service!");</literal_exp>
        <retType_exp id="3">
          <literal_exp>_tab</literal_exp>
          <literal_exp>return </literal_exp>
          <retValue/>
          <literal_exp>;</literal_exp>
        </retType_exp>
        <literal_exp>}</literal_exp>
        <literal_exp>}</literal_exp>
      </operation_exp>
      <literal_exp>_newline</literal_exp>
      <literal_exp>}</literal_exp>
  </element_exp>
</pattern>
```

Listing 5.3: Generation of Java RMI-specific Illusion Capsules

```
<?xml version="1.0"?>
<pattern type="Capsule" platform="RMI">
  <reference_exp>
    <literal_exp>import java.rmi.*;</literal_exp>
    <literal_exp>import java.rmi.server.*;</literal_exp>
    <literal_exp>import java.net.*;</literal_exp>
    <literal_exp>public class </literal_exp>
    <reference_name/>
    <literal_exp>Server {</literal_exp>
    <literal_exp>public static void main(String args[]) {</literal_exp>
    <literal_exp>try {</literal_exp>
    <element_name/>
    <literal_exp> impl = new </literal_exp>
    <element_name/>
    <literal_exp>("http://</literal_exp>
    <IPAddress/>
    <literal_exp>:8080/axis/</literal_exp>
    <jws_exp>
      <jws id="1">
        <literal_exp>WS_</literal_exp>
        <reference_name/>
        <literal_exp>_</literal_exp>
        <server_platform/>
        <literal_exp>Client</literal_exp>
      </jws>
      <jws id="2">
        <reference_name/>
        <literal_exp>Server</literal_exp>
      </jws>
    </jws_exp>
    <literal_exp>.jws");</literal_exp>
    <literal_exp>Naming.rebind("</literal_exp>
    <reference_name/>
    <literal_exp>", impl);</literal_exp>
    <literal_exp>System.out.println("RMI </literal_exp>
    <reference_name/>
    <literal_exp> started on IP " + InetAddress.getLocalHost().
       getHostAddress()+"\n" );</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>catch(Exception ex) {</literal_exp>
    <literal_exp>System.err.println("ERROR: " + ex);</literal_exp>
    <literal_exp>ex.printStackTrace(System.out);</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>}</literal_exp>
  </reference_exp>
</pattern>
```

WebServices-to-CORBA pattern

Listing 5.4: Generation of Web Services that serve as CORBA Clients

```
<?xml version="1.0"?>
<pattern type="ObjectImpl" platform="WS">
  <datatypes>
    <datatype id="0" name="void" retValue="null">void</datatype>
                <datatype id="1" name="boolean" retValue="false" holder="
                    javax.xml.rpc.holders.BooleanHolder">boolean</datatype>
                <datatype id="2" name="char" retValue="'\u0000'" holder="
                    javax.xml.rpc.holders.StringHolder">char</datatype>
                <datatype id="3" name="string" retValue="null" holder="
                    javax.xml.rpc.holders.StringHolder">String</datatype>
                <datatype id="4" name="byte" retValue="-1" holder="javax.
                    xml.rpc.holders.ByteHolder">byte</datatype>
                <datatype id="5" name="unsigned byte" retValue="-1" holder=
                    "javax.xml.rpc.holders.ShortHolder">short</datatype>
                <datatype id="6" name="short" retValue="-1" holder="javax.
                    xml.rpc.holders.ShortHolder">short</datatype>
                <datatype id="7" name="unsigned short" retValue="-1" holder
                    ="javax.xml.rpc.holders.ShortHolder">short</datatype>
                <datatype id="8" name="int" retValue="-1" holder="javax.xml
                    .rpc.holders.IntHolder">int</datatype>
                <datatype id="9" name="unsigned int" retValue="-1" holder="
                    javax.xml.rpc.holders.IntHolder">int</datatype>
                <datatype id="10" name="long" retValue="-1" holder="javax.
                    xml.rpc.holders.LongHolder">long</datatype>
                <datatype id="11" name="unsigned long" retValue="-1" holder
                    ="javax.xml.rpc.holders.LongHolder">long</datatype>
                <datatype id="12" name="float" retValue="-1.0" holder="
                    javax.xml.rpc.holders.FloatHolder">float</datatype>
                <datatype id="13" name="double" retValue="-1.0" holder="
                    javax.xml.rpc.holders.DoubleHolder">double</datatype>
                <datatype id="14" name="BigInt" retValue="java.math.
                    BigInteger.ZERO" holder="javax.xml.rpc.holders.
                    BigIntegerHolder">java.math.BigInteger</datatype>
                <datatype id="15" name="decimal" retValue="new java.math.
                    BigDecimal(java.math.BigInteger.ZERO)" holder="javax.xml
                    .rpc.holders.BigDecimalHolder">java.math.BigDecimal</
                    datatype>
  </datatypes>
  <wrapper_exp>
    <literal_exp>import org.omg.CosNaming.*;</literal_exp>
                <literal_exp>import org.omg.CosNaming.NamingContextPackage
                    .*;</literal_exp>
                <literal_exp>import org.omg.CORBA.*;</literal_exp>
                <literal_exp>import java.net.*;</literal_exp>
                <literal_exp>public class WS_</literal_exp>
                <reference_name/>
                <literal_exp>_</literal_exp>
                <literal_exp>CorbaClient {</literal_exp>
                <interf_name/>
                <literal_exp> ref;</literal_exp>
  <wrapperoperation_exp>
    <literal_exp>public </literal_exp>
    <retType/>
                        <opName/>
                        <literal_exp>(</literal_exp>
    <arg_exp>
      <direction_type_argName/>
      <literal_exp>,</literal_exp>
    </arg_exp>
    <literal_exp>{</literal_exp>
                        <literal_exp>try {</literal_exp>
                        <literal_exp>String [] args = {"-ORBInitialPort","
                            1050","-ORBInitialHost", InetAddress.
                            getLocalHost().getHostAddress()};</literal_exp>
                        <literal_exp>ORB orb = ORB.init(args, null);</
                            literal_exp>
                        <literal_exp>org.omg.CORBA.Object objRef = orb.
                            resolve_initial_references("NameService");</
                            literal_exp>
                        <literal_exp>NamingContextExt ncRef =
                            NamingContextExtHelper.narrow(objRef);</
                            literal_exp>
                        <literal_exp>ref = </literal_exp>
                        <interf_name/>
                        <literal_exp>Helper.narrow(ncRef.resolve_str("</
                            literal_exp>
                        <reference_name/>
                        <literal_exp>"));</literal_exp>
                        <literal_exp>System.out.println("Obtain a handle on
                             Corba </literal_exp>
    <reference_name/>
    <literal_exp> server object");</literal_exp>
    <invoke>
      <retType_exp id="1">
        <retType/>
        <literal_exp> ret = (</literal_exp>
        <retType/>
        <literal_exp>) </literal_exp>
      </retType_exp>
      <literal_exp>ref.</literal_exp>
      <opName/>
      <literal_exp>(</literal_exp>
      <arg_exp>
        <direction_type_argName/>
        <literal_exp>,</literal_exp>
      </arg_exp>
```

```
          <literal_exp>);</literal_exp>
          <retType_exp id="2">
            <literal_exp>_tab</literal_exp>
            <literal_exp>return ret;</literal_exp>
          </retType_exp>
        </invoke>
        <literal_exp>}</literal_exp>
        <literal_exp>catch(Exception ex) {</literal_exp>
        <literal_exp>ex.printStackTrace();</literal_exp>
        <literal_exp>System.out.println("Cannot establish connection with
           Corba server!");</literal_exp>
        <literal_exp>System.exit(0);</literal_exp>
        <retType_exp id="3">
          <literal_exp>_tab</literal_exp>
          <literal_exp>return </literal_exp>
          <retValue/>
          <literal_exp>;</literal_exp>
        </retType_exp>
        <literal_exp>}</literal_exp>
        <literal_exp>}</literal_exp>
      </wrapperoperation_exp>
      <literal_exp>}</literal_exp>
  </wrapper_exp>
</pattern>
```

CORBA-to-RMI pattern

To generate CORBA illusion interfaces, we can use the corresponding part of the CORBA-to-WebServices pattern (see listing 3.3).

Listing 5.5: Generation of CORBA-specific Object Implementations that serve as Java RMI Clients

```
<?xml version="1.0"?>
<pattern type="ObjectImpl" platform="Corba">
  <datatypes>
    <datatype id="0" name="void">void</datatype>
    <datatype id="1" name="boolean" holder="BooleanHolder" retValue="false"
        >boolean</datatype>
                <datatype id="2" name="char" holder="CharHolder" retValue="
                    '\u0000'">char</datatype>
                <datatype id="3" name="string" holder="StringHolder"
                    retValue="null">String</datatype>
                <datatype id="4" name="byte" holder="ByteHolder" retValue="
                    -1">byte</datatype>
                <datatype id="5" name="unsigned byte" holder="ShortHolder"
                    retValue="-1">short</datatype>
                <datatype id="6" name="short" holder="ShortHolder" retValue
                    ="-1">short</datatype>
                <datatype id="7" name="unsigned short" holder="ShortHolder"
                     retValue="-1">short</datatype>
                <datatype id="8" name="int" holder="IntHolder" retValue="-1
                    ">int</datatype>
                <datatype id="9" name="unsigned int" holder="IntHolder"
                    retValue="-1">int</datatype>
                <datatype id="10" name="long" holder="LongHolder" retValue=
                    "-1">long</datatype>
                <datatype id="11" name="unsigned long" holder="LongHolder"
                    retValue="-1">long</datatype>
                <datatype id="12" name="float" holder="FloatHolder"
                    retValue="-1.0">float</datatype>
                <datatype id="13" name="double" holder="DoubleHolder"
                    retValue="-1.0">double</datatype>
                <datatype id="14" name="BigInt" holder="ObjectHolder"
                    retValue="java.math.BigInteger.ZERO">java.math.
                    BigInteger</datatype>
                <datatype id="15" name="decimal" holder="ObjectHolder"
                    retValue="new java.math.BigDecimal(java.math.BigInteger.
                    ZERO)">java.math.BigDecimal</datatype>
  </datatypes>
  <element_exp>
    <literal_exp>import org.omg.CosNaming.*;</literal_exp>
    <literal_exp>import org.omg.CosNaming.NamingContextPackage.*;</
        literal_exp>
    <literal_exp>import org.omg.CORBA.*;</literal_exp>
    <literal_exp>import org.omg.PortableServer.*;</literal_exp>
    <literal_exp>import org.omg.PortableServer.POA;</literal_exp>
    <literal_exp>import java.rmi.*;</literal_exp>
    <literal_exp>import java.rmi.server.*;</literal_exp>
    <literal_exp>import java.util.Properties;</literal_exp>
    <literal_exp>import java.net.*;</literal_exp>
    <literal_exp>import java.lang.*;</literal_exp>
    <literal_exp>public class </literal_exp>
    <element_name/>
    <literal_exp> extends </literal_exp>
    <interf_name/>
    <literal_exp>POA { </literal_exp>
    <literal_exp>private ORB orb;</literal_exp>
    <literal_exp>private String RMIServerURL;</literal_exp>
    <interf_name/>
    <literal_exp> ref;</literal_exp>
                <literal_exp>public </literal_exp>
    <element_name/>
    <literal_exp>(String servURL) {</literal_exp>
    <literal_exp>RMIServerURL = servURL;</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>public void setORB(ORB orb_val) {</literal_exp>
    <literal_exp>this.orb = orb_val;</literal_exp>
    <literal_exp>}</literal_exp>
    <operation_exp>
      <literal_exp>public </literal_exp>
      <retType/>
      <opName/>
      <literal_exp>(</literal_exp>
      <arg_exp>
        <direction_type_argName/>
        <literal_exp>,</literal_exp>
      </arg_exp>
      <literal_exp>) {</literal_exp>
      <literal_exp>try {</literal_exp>
      <literal_exp>ref = (</literal_exp>
      <interf_name/>
      <literal_exp>) Naming.lookup(RMIServerURL);</literal_exp>
      <literal_exp>System.out.println("Obtain a handle on RMI </literal_exp
         >
      <reference_name/>
      <literal_exp> server object");</literal_exp>
      <invoke>
        <retType_exp id="1">
          <retType/>
          <literal_exp> ret = (</literal_exp>
          <retType/>
          <literal_exp>) </literal_exp>
        </retType_exp>
        <literal_exp>ref.</literal_exp>
        <opName/>
        <literal_exp>(</literal_exp>
        <arg_exp>
          <direction_type_argName/>
          <literal_exp>,</literal_exp>
        </arg_exp>
        <literal_exp>);</literal_exp>
        <retType_exp id="2">
          <literal_exp>return ret;</literal_exp>
        </retType_exp>
      </invoke>
      <literal_exp>}</literal_exp>
      <literal_exp>catch(Exception ex) {</literal_exp>
      <literal_exp>ex.printStackTrace();</literal_exp>
      <literal_exp>System.out.println("Cannot establish connection with RMI
           server!");</literal_exp>
      <literal_exp>System.exit(0);</literal_exp>
      <retType_exp id="3">
        <literal_exp>return </literal_exp>
        <retValue/>
        <literal_exp>;</literal_exp>
      </retType_exp>
      <literal_exp>}</literal_exp>
      <literal_exp>}</literal_exp>
    </operation_exp>
    <literal_exp>}</literal_exp>
  </element_exp>
</pattern>
```

Listing 5.6: Generation of CORBA-specific Illusion Capsules that host CORBA Objects serving as Java RMI Clients

```
<?xml version="1.0"?>
<pattern type="Capsule" platform="Corba">
  <reference_exp>
    <literal_exp>import org.omg.CosNaming.*;</literal_exp>
    <literal_exp>import org.omg.CosNaming.NamingContextPackage.*;</
        literal_exp>
    <literal_exp>import org.omg.CORBA.*;</literal_exp>
                <literal_exp>import org.omg.PortableServer.*;</literal_exp>
    <literal_exp>import org.omg.PortableServer.POA;</literal_exp>
    <literal_exp>import java.net.*;</literal_exp>
    <literal_exp>public class </literal_exp>
    <reference_name/>
    <literal_exp>Server {</literal_exp>
    <literal_exp>public static void main(String args[]) {</literal_exp>
    <literal_exp>try {</literal_exp>
    <literal_exp>ORB orb = ORB.init(args, null);</literal_exp>
    <literal_exp>POA rootpoa = POAHelper.narrow(orb.
        resolve_initial_references("RootPOA"));</literal_exp>
    <literal_exp>rootpoa.the_POAManager().activate();</literal_exp>
    <literal_exp>org.omg.CORBA.Object objRef = orb.
        resolve_initial_references("NameService");</literal_exp>
    <literal_exp>NamingContextExt ncRef = NamingContextExtHelper.narrow(
        objRef);</literal_exp>
    <element_name/>
    <literal_exp> impl = new </literal_exp>
    <element_name/>
    <literal_exp>("rmi://<!--"+ InetAddress.getLocalHost().getHostAddress()
        +"/--></literal_exp>
    <IPAddress/>
    <literal_exp>/</literal_exp>
    <reference_name/>
    <literal_exp>");</literal_exp>
    <literal_exp>impl.setORB(orb);</literal_exp>
    <literal_exp>org.omg.CORBA.Object ref = rootpoa.servant_to_reference(
        impl);</literal_exp>
    <interf_name/>
    <literal_exp> href = </literal_exp>
    <interf_name/>
    <literal_exp>Helper.narrow(ref);</literal_exp>
    <literal_exp>NameComponent path[] = ncRef.to_name("</literal_exp>
    <reference_name/>
    <literal_exp>");</literal_exp>
    <literal_exp>ncRef.rebind(path, href);</literal_exp>
    <literal_exp>System.out.println("Corba </literal_exp>
    <reference_name/>
    <literal_exp> started on IP " + InetAddress.getLocalHost().
       getHostAddress()+"\n" );</literal_exp>
    <literal_exp>orb.run();</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>catch(Exception ex) {</literal_exp>
    <literal_exp>System.err.println("ERROR: " + ex);</literal_exp>
    <literal_exp>ex.printStackTrace(System.out);</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>}</literal_exp>
  </reference_exp>
</pattern>
```

RMI-to-CORBA pattern

To generate Java RMI illusion interfaces, we can use the corresponding part of the RMI-to-WebServices pattern (see listing 5.1 in the current appendix).

Listing 5.7: Generation of Java RMI-specific Object Implementations that serve as CORBA Clients

```
<?xml version="1.0"?>
<pattern type="ObjectImpl" platform="RMI">
  <datatypes>
    <datatype id="0" name="void" retValue="null">void</datatype>
                <datatype id="1" name="boolean" retValue="false">boolean</
                    datatype>
                <datatype id="2" name="char" retValue="'\u0000'">char</
                    datatype>
                <datatype id="3" name="string" retValue="null">String</
                    datatype>
                <datatype id="4" name="byte" retValue="-1">byte</datatype>
                <datatype id="5" name="unsigned byte" retValue="-1">short</
                    datatype>
                <datatype id="6" name="short" retValue="-1">short</datatype
                    >
                <datatype id="7" name="unsigned short" retValue="-1">short<
                    /datatype>
                <datatype id="8" name="int" retValue="-1">int</datatype>
                <datatype id="9" name="unsigned int" retValue="-1">int</
                    datatype>
                <datatype id="10" name="long" retValue="-1">long</datatype>
                <datatype id="11" name="unsigned long" retValue="-1">long</
                    datatype>
                <datatype id="12" name="float" retValue="-1.0">float</
                    datatype>
                <datatype id="13" name="double" retValue="-1.0">double</
                    datatype>
                <datatype id="14" name="BigInt" retValue="java.math.
                    BigInteger.ZERO">java.math.BigInteger</datatype>
                <datatype id="15" name="decimal" retValue="new java.math.
                    BigDecimal(java.math.BigInteger.ZERO)">java.math.
                    BigDecimal</datatype>
  </datatypes>
  <element_exp>
    <literal_exp>import java.rmi.*;</literal_exp>
    <literal_exp>import java.rmi.server.*;</literal_exp>
    <literal_exp>import org.omg.CosNaming.*;</literal_exp>
    <literal_exp>import org.omg.CosNaming.NamingContextPackage.*;</
        literal_exp>
    <literal_exp>import org.omg.CORBA.*;</literal_exp>
    <literal_exp>import java.util.Properties;</literal_exp>
```

```
<literal_exp>import java.net.*;</literal_exp>
<literal_exp>import java.lang.*;</literal_exp>
<literal_exp>public class </literal_exp>
<element_name/>
<literal_exp> extends UnicastRemoteObject implements </literal_exp>
<interf_name/>
<literal_exp> {</literal_exp>
<interf_name/>
<literal_exp> ref;</literal_exp>
<literal_exp>private String []args;</literal_exp>
            <literal_exp>public </literal_exp>
<element_name/>
<literal_exp>() throws RemoteException {</literal_exp>
<literal_exp>args = new String[4];</literal_exp>
<literal_exp>args[0] = "-ORBInitialPort";</literal_exp>
<literal_exp>args[1] = "1050";</literal_exp>
<literal_exp>args[2] = "-ORBInitialHost";</literal_exp>
<literal_exp>args[3] = "</literal_exp>
<IPAddress/>
<literal_exp>";</literal_exp>
<literal_exp>}</literal_exp>
<operation_exp>
  <literal_exp>public </literal_exp>
  <retType/>
  <opName/>
  <literal_exp>(</literal_exp>
  <arg_exp>
    <direction_type_argName/>
    <literal_exp>,</literal_exp>
  </arg_exp>
  <literal_exp>) {</literal_exp>
  <literal_exp>try {</literal_exp>
  <literal_exp>ORB orb = ORB.init(args,null);</literal_exp>
  <literal_exp>org.omg.CORBA.Object objRef = orb.
      resolve_initial_references("NameService");</literal_exp>
  <literal_exp>NamingContextExt ncRef = NamingContextExtHelper.narrow(
      objRef);</literal_exp>
  <literal_exp>ref = </literal_exp>
  <interf_name/>
  <literal_exp>Helper.narrow(ncRef.resolve_str("</literal_exp>
  <reference_name/>
  <literal_exp>"));</literal_exp>
  <literal_exp>System.out.println("Obtain a handle on Corba </
      literal_exp>
  <reference_name/>
  <literal_exp> server object");</literal_exp>
  <invoke>
    <retType_exp id="1">
      <retType/>
      <literal_exp> ret = (</literal_exp>
          <retType/>
          <literal_exp>) </literal_exp>
        </retType_exp>
        <literal_exp>ref.</literal_exp>
        <opName/>
        <literal_exp>(</literal_exp>
        <arg_exp>
          <direction_type_argName/>
          <literal_exp>,</literal_exp>
        </arg_exp>
        <literal_exp>);</literal_exp>
        <retType_exp id="2">
          <literal_exp>return ret;</literal_exp>
        </retType_exp>
      </invoke>
      <literal_exp>}</literal_exp>
      <literal_exp>catch(Exception ex) {</literal_exp>
      <literal_exp>ex.printStackTrace();</literal_exp>
      <literal_exp>System.out.println("Cannot establish connection with
         Corba server!");</literal_exp>
      <literal_exp>System.exit(0);</literal_exp>
      <retType_exp id="3">
        <literal_exp>return </literal_exp>
        <retValue/>
        <literal_exp>;</literal_exp>
      </retType_exp>
      <literal_exp>}</literal_exp>
      <literal_exp>}</literal_exp>
    </operation_exp>
    <literal_exp>}</literal_exp>
  </element_exp>
</pattern>
```

Listing 5.8: Generation of Java RMI-specific Illusion Capsules that host Java Objects serving as CORBA Clients

```
<?xml version="1.0"?>
<pattern type="Capsule" platform="RMI">
  <reference_exp>
    <literal_exp>import java.rmi.*;</literal_exp>
    <literal_exp>import java.rmi.server.*;</literal_exp>
    <literal_exp>import java.net.*;</literal_exp>
    <literal_exp>public class </literal_exp>
    <reference_name/>
    <literal_exp>Server {</literal_exp>
    <literal_exp>public static void main(String args[]) {</literal_exp>
    <literal_exp>try {</literal_exp>
    <element_name/>
    <literal_exp> impl = new </literal_exp>
    <element_name/>
    <literal_exp>();</literal_exp>
    <literal_exp>Naming.rebind("</literal_exp>
    <reference_name/>
    <literal_exp>", impl);</literal_exp>
    <literal_exp>System.out.println("RMI </literal_exp>
    <reference_name/>
    <literal_exp> started on IP " + InetAddress.getLocalHost().
       getHostAddress()+"\n" );</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>catch(Exception ex) {</literal_exp>
    <literal_exp>System.err.println("ERROR: " + ex);</literal_exp>
    <literal_exp>ex.printStackTrace(System.out);</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>}</literal_exp>
    <literal_exp>}</literal_exp>
  </reference_exp>
</pattern>
```

Short CV

Iason Tsaparlis was born in Ioannina in 1982. He completed the high school in Ioannina in 2000, and obtained his Diploma in Computer Engineering from the Department of Computer Engineering and Informatics of the University of Patras in 2005. The theme of his undergraduate diploma thesis was ”Implementation of a Multilingual Electronic Magazine by using XML Technology”. After fulfilling his national service in the Greek army, he started his postgraduate studies in the Department of Computer Science of the University of Ioannina, in 2007. His research focused on the problem of middleware platform heterogeneity and interoperability. He is a member of the Technical Chamber of Greece (TEE) since 2006.