Universidad Politécnica de Valencia

Departamento de Sistemas Informáticos y Computación

A Compiler for the Automatic Generation Of the Distribution Aspect To Distributed Aplications

"COMPILADOR PARA LA GENERACIÓN AUTOMÁTICA DEL ASPECTO DE DISTRIBUCION A APLICACIONES DISTRIBUIDAS"

Programa de Doctorado: **Programación Declarativa e Ingeniería de la Programación**

Nour Ali

Dirigido por: Dr. Isidro Ramos Salavert
Dr. Jose Ángel Carsí Cubel

Valencia, Septiembre 2004

ABSTRACT

This research work presents a manual compiler for generating distributed and mobile applications from the PRISMA distribution model. The MDA proposal is applied to our PRISMA distribution Model. The steps of the proposal have been followed. First, a Platform Specific Model (PSM) of the PRISMA distribution model in C# .Net is presented. The PSM is presented by discussing the Attachments, Bindings and Mobility Specific Model in C# by showing their logical diagrams. Then, the implementation patterns for generating distributed application in C# from the PRISMA distribution model are identified. At this level, the PRISMA distribution model is implemented in C#. Some predefined classes that form the middleware have been necessary to create PRISMA distribution applications.

Keywords: distributed systems, architectural models, aspect-oriented software development (AOSD), automatic code generation, MDA, transformation patterns, middlewares.

ACKNOWLEDGMENTS

To my parents, for their love, care and encouragement.
To my brother and sisters, Mohamed, Majd and Dana, and to all my Family

To Isidro, for accepting me as a student,
To Pepe, for his comments and advisments

To Jenny for being a marvellous work partner and most of all,
For being a friend, You are very special, I am glad to meet you.

To the best programming guys,
Jose, Cristobal and Rafa, Keep working hard!

To Microsoft Research,
For financing this work.

To my friends, I miss you so much.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Nowadays, information systems are large and complex to develop. An important factor that influences in this complexity is that information systems are tending to be distributed with mobile components. Many technologies have emerged in dealing with distribution issues at an implementation level. On the other hand, few approaches have dealt with distribution at a high abstraction level. Nevertheless, considering distribution in the whole life cycle of software development minimizes time and costs. Thus, efforts are reduced in the development process by taking into account distribution at an early phase, instead of only introducing it at the implementation phase.

Actually, many CASE tools are able to generate applications following the automatic prototyping paradigm which was proposed by Balzer [6]. They are called model compilers and are able to automatically generate applications from conceptual models. This generation can be a complete generation as in Oblog Case [23] and Sosy Modeler® (OO-Method/CASE [18]), or partial, like Rational Rose [22]. Specifically, Sosy Modeler® (OO-Method/CASE) generates aplications with a three layered architecture (presentation, business logic and persistence) and uses OASIS as a specification language (See Figure 1). During the last decade, the research efforts were dedicated to formalize the models to automatically generate applications mainly using the object oriented paradigm such as Oblog [17] or Troll [14].

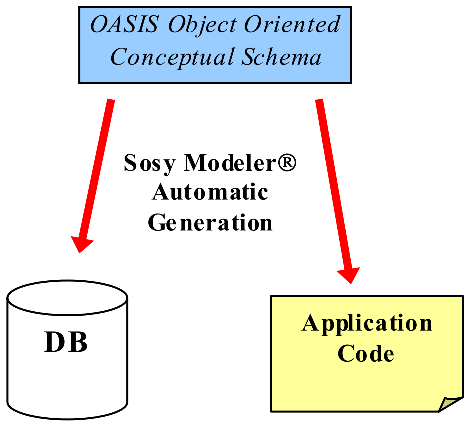

OASIS Object Oriented Conceptual Schema
Sosy Modeler® Automatic Generation
DB
Application Code

Figure 1 Sosy Modeler® Automatic Generation

As a consequence of the poor capability of the object-oriented software development to describe complex structures of distributed information systems emerged the Component-Based Software Development (CBSD) [26] [4] and the Aspect-Oriented Software Development (AOSD) [4] have emerged. The CBSD promises to control the complexity of system construction by coupling entities that provide specific services. The AOSD allows separation of concerns by modularizing crosscutting concerns in a separate entity: the aspect. Aspects can be reused and manipulated independently of the rest of properties of the system. However, currently no model compilers exist that combine the CBSD and AOSD to generate distributed applications.

In previous works as in [2] [3][1], a distribution model for the architectural model PRISMA [19] was proposed using the AOSD and CBSD. This work, presents the implementation of this model in C# .Net Remoting [21] and some code generation patterns.

In this chapter, the objectives of the research work are introduced in Section 1.1, and the structure of the document is presented in Section 1.2.

1.1 Research Objectives

The objective of this research work is to identify the transformation patterns that allow the automatic code generation of C# .Net from the PRISMA distribution model previously presented in other works [1] [3] .

The MDA proposal [15] [24] of identifying transformation patterns (technology patterns and implementation patterns) for generating code from a platform independent model (PIM) is used. First the platform specific PRISMA distribution model in C# is going to be described by showing the logical diagrams that form it. Then the implementation patterns that indicate which classes of the platform specific model are to be extended to generate distributed C# applications of the PRISMA distribution model are identified.

In the future, the PRISMA architectural model is pretended to become a framework that permits the automatic generation of distributed information systems.

1.2 Structure of the Document

This work is divided into four chapters and two appendixes. In the following the content of each of the chapters is briefly described:

Chapter 2 presents some fundamentals and related work to ours. The fundamentals presented in this chapter are necessary to understand the research work. The MDA proposal, the concept of a middleware, framework and mobility are presented. In addition, a comparison between ASP.Net Web Services and .Net Remoting is presented.

Chapter 3 introduces how the MDA proposal is applied to our PRISMA distribution Model. The steps of the proposal have been followed. First, a Platform Specific Model (PSM) of the PRISMA distribution model in C# .Net is presented. The PSM is presented by discussing the Attachments, Bindings and Mobility Specific Model in C# by showing their logical diagrams. Then, the implementation patterns for generating distributed application in C# from the PRISMA distribution model are identified. At this level, the PRISMA distribution model is implemented in C#. Some predefined classes that form the middleware have been necessary to create PRISMA distribution applications.

Chapter 4 sums up the main contributions of the work and suggest some future works.

Appendix A shows the mappings of the whole PRISMA architectural model to C#.Net. This appendix has been included to facilitate to the user the readability of the document.

Appendix B shows the code of an implemented Bank system example in C# using the PRISMA distribution model.

CHAPTER 2. FUNDAMENTALS AND RELATED WORK

2.1 Model Driven Architectures (MDA)

Nowadays, the software life cycle is very complex, facing a problem named the integration problem. This problem occurs due to that information systems are very large and need to integrate many tools and middlewares. Model Driven Architecture (MDA) [15][24] is an Object Management Group (OMG) standard for model-based application architectures. MDA tries to solve the integration problem encountered in today's development process (see Figure 2). MDA defines an approach to information technology specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. To this end, the MDA defines an architecture for models that provides a set of guidelines for structuring specifications expressed as models.

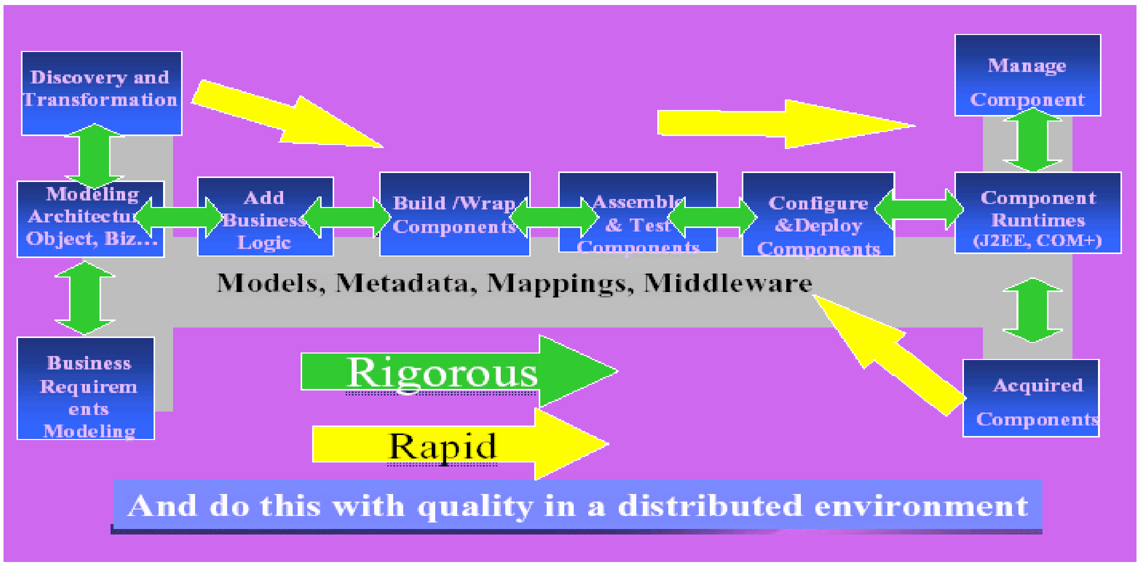

Discovery and Transformation
Manage Component
Modeling Architecture Object, Biz...
Add Business Logic
Build /Wrap Components
Assemble & Test Components
Configure &Deploy Components
Component Runtimes (J2EE, COM+)
Models, Metadata, Mappings, Middleware
Business Requirements Modeling
Rigorous
Rapid
Acquired Components
And do this with quality in a distributed environment

Figure 2 Complex Life Cycle

MDA (see Figure 3) is a software architecture based on modelling at different levels. Its core is a technology-independent definition of the distributed enterprise infrastructure built in OMG's Unified Modelling Language (UML) [29]. Using the UML model the running application is automatically generated. The MDA promises:

-To improve productivity for architects

-To integrate what you've built, with what you're building, with what you will build in the future;

-To enhance portability and Interoperability.

-To remain flexible in the face of constantly changing infrastructure;

-To lengthen the usable lifetime of your software, lowering maintenance costs

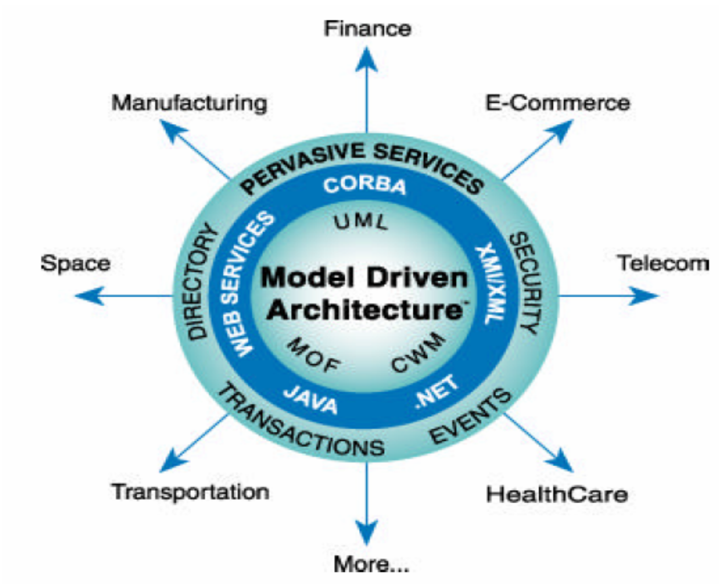


Figure 3 OMG's Model Driven Architecture

MDA based development enables to produce applications with different middleware platforms from the same base model. All MDA development projects start with the creation of a Platform Independent Model (PIM) (see Figure 4). The PIM expresses an application's fundamental business functionality and behaviour in a platform independent way. Then from the same PIM the definition of a Platform Specific Model (PSM) for a certain target platform. The PSM defines an application, based on a certain technology, but without any coding detail. The PSM describes what is to be generated to implement the application. Next, as shown in Figure 4 the code model defines the application actually implemented in code.

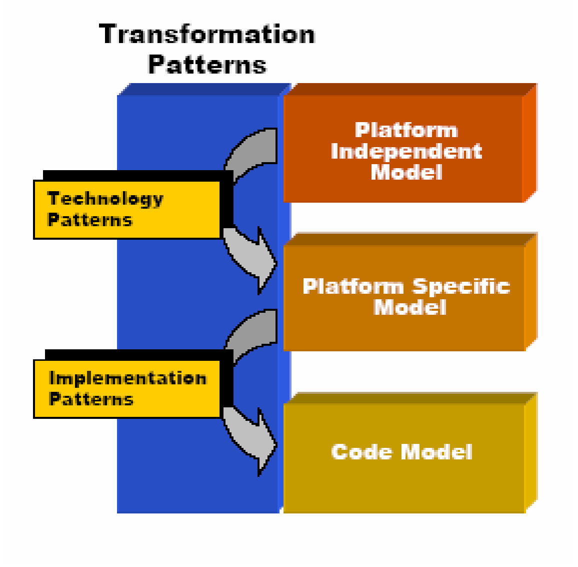


Figure 4 Model Transformations in MDA

To transform one model to elements in a lower model, transformation patterns are used. Transformation patterns in MDA are classified into two categories: Technology Patterns and Implementation Patterns. Technology Patterns map PIM's to PSM's for a specific technology and application architecture. Implementation Patterns map PSM´s specifications to code.

An example of an MDA-based development environment is OptimalJ [10]. OptimalJ enables the design, development, modification and deployment of J2EE business applications. It blends models, patterns and application frameworks. OptimalJ generates working applications directly from a visual model, using active synchronization to keep both model and code synchronously during application development.

2.2 Middleware

A middleware is an intermediate layer that sits above a platform and below the applications (see Figure 5). A platform is the set of services of an operating system and networking programmes. This layer provides services which help in providing development and run-time environment for distributed systems by making distribution transparent and solving heterogeneity such as hardware, programming language and operating systems. These middleware services are general purpose services defined by the application programming interfaces (API's) and protocols they support. They may have multiple implementations that conform to their interfaces and protocols.

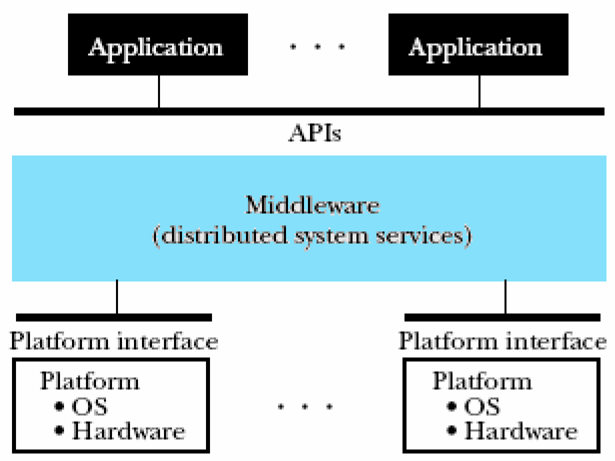


Figure 5 Middleware

Bernstein [7] describes properties for distinguishing middleware services from non-middleware services. He describes middleware services as generic distributed services across applications and industries that run on multiple platforms and support standard interfaces and protocols. Middleware services are generic applications meeting the need of a variety of applications across many industries having implementations that run on multiple platforms.

Thus, a middleware service should support a standard protocol such as TCP/IP. That way, multiple implementations of the service can be developed and those implementations will interoperate. Middleware services must be accessed remotely or enable others to be accessed remotely. A middleware service is transparent with respect to an API to allow applications to use the API without modification.

Middleware are classified into different kinds to serve different purposes. Some of these kinds are the reflective, object-oriented, message-oriented and event-based middleware. The reflective middleware is concerned with applying techniques from the field of reflection in order to achieve flexibility and adaptability in middleware platforms. Event-based middleware is concerned with the concepts, design, implementation, and application of services and components that support building event-based systems. Object-oriented middleware extends the object-oriented programming paradigm to distributed systems. Message-oriented middleware is the natural extension of the packet paradigm of communications prevalent in the lower layers of the OSI network model.

2.3 Framework

A framework is defined by Bernstein [7] as a software environment that is designed to simplify application development and system management for a specialized domain. Bernstein describes frameworks as kinds of abstract middlewares which sit on the middleware to simplify their underlying middleware environment rather than directly accessing middleware services (see Figure 6).

A framework is defined by an API, a user interface, a set of tools and may also have framework-private services in addition to the ones that middleware environments offer. A framework API simplifies the API’s of the underlying middleware. A framework API could be simply an abstracted middleware API that maps the framework services to services of the middleware or it could be different. When a framework API is different it can add framework-private middleware services.

A framework includes tools which make the framework easier to use. Tools could be editors, compilers, help facilities and software installation managers. A tool is part of the framework if it is integrated through data. Mostly, frameworks contain a repository that is shared by tools for analysis or design such as CASE frameworks.

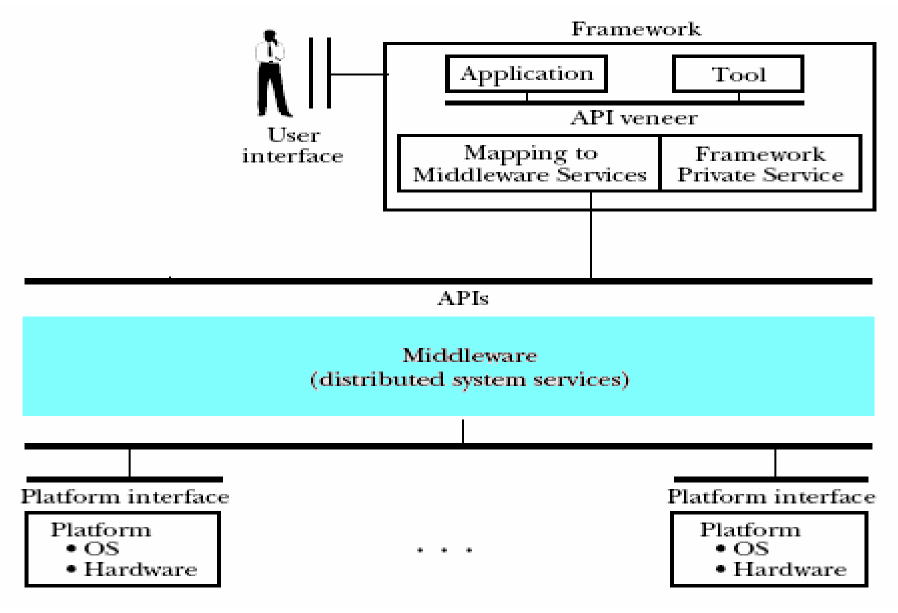


Figure 6 Framework Architecture

2.4 Mobility

Mobility is the capability of moving a process, object or component instances from one computing node to another during the runtime in a distributed system.

Mobility is classified by Picco [13] into weak and strong mobility. Weak mobility happens in systems where the migrant is a data object which starts execution from the beginning after migration. Weak mobility transfers the code which may be accompanied by some initialization data, however the state is not involved. This kind of migration is well known in commercial systems. Strong mobility occurs in mobile objects which their execution is interrupted for the migration and once migrated on the destination carries forward executing from the interrupted point. This form of mobility allows migration of both the code and the state of the object before interrupting it.

Strong Mobility is supported by two mechanisms: migration and cloning. The migration mechanism destroys the executing object and transmits it to the destination. Migration can be proactive and reactive. In proactive migration, the decision of moving the object is done by itself determining the time and destination. While in reactive migration the migration decision is determined by another executing object. The cloning mechanism creates a copy of the executing object at the new destination without destroying the executing object. As in migration, cloning can be proactive and reactive.

Weak mobility's mechanisms are influenced on the direction of code transfer, the type of code and the time the code is executed at the destination. The code can be migrated as a standalone or as a code fragment. Standalone code is self-contained and will be used to instantiate a new object on the destination. A code fragment must be connected to an already running code. Mechanisms that support weak mobility can be either synchronous and asynchronous. Figure 7, summarizes Picco's classification of migration.

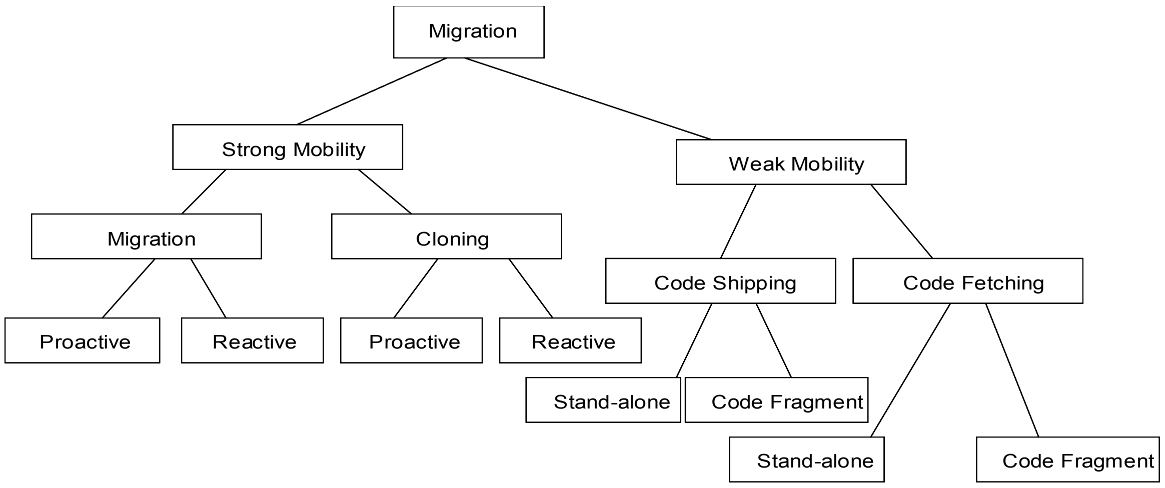


Figure 7 Picco's classification of Migration

Migration is not totally supported by today's middleware technologies. Therefore, different proposals have been done to extend the frameworks. The work in [27] describes the integration of a migration facility into the .Net Framework. Using aspect-techniques for integrating migration into .NET addresses non-functional system properties on the middleware level, without the need to manipulate lower system layers like the operating system itself.

2.5 Distributed Technologies.

Nowadays, distributed systems are built using distributed object or component middleware. The role of middleware is to ease the task of programming and managing distributed applications. It is a is a distributed software layer, or 'platform' which abstracts over the complexity and heterogeneity of the underlying distributed environment with its multitude of network technologies, machine architectures, operating systems and programming languages.

In the following a brief explanation of the most recent proposals using middlewares are presented:

- **OMG's CORBA** [9] is an object based middleware which offers an interface definition language (IDL) and an object request broker (ORB). The IDL specifies the interfaces among the CORBA objects. The IDL is used to abstract over the fact that objects can be implemented in any suitable programming language. In addition, the IDL is responsible to ensure that data is correctly interchanged among the different programming languages. The ORB is responsible for transparently directing method invocations to the appropriate target object, and a set of *services* (e.g. naming, time, transactions, replication etc.)

- **Microsoft® .NET Remoting** [16][21] provides a framework that allows objects to interact with one another across application domains. The framework provides a number of services, including activation and lifetime support, as well as communication channels responsible for transporting messages to and from remote applications. The framework can be extended to achieve what is required.
- **Web Services** [30] typically use SOAP for the message format and require that you use IIS for the HTTP message transport. This makes Web services good for communication over the Internet, and for communication between non-Windows systems. Web services are a good choice for message-oriented services that must support a wide range of client platforms and a potentially heavy load.

2.5.1 Comparison between ASP.Net Web Services and .Net Remoting

This section is essential for this work to determine which most adequate technology to choose to implement our model. We are interested in implementing our model in a .Net technology. Therefore, we have to choose either ASP.Net Web Services or .Net Remoting. To do this study we have depended on the following works: [11], [12] , [25] and [28].

The .Net Remoting Architecture consists of the proxy, formatter and a transport channel on each application domain (see Figure 8). The proxy receives the call from a client, encodes the message using an appropriate formatter, then sends the call over the channel to the server process. A listening channel on the server application domain picks up the request and forwards it to the server remoting system, which locates and invokes the methods on the requested object. Once the execution is completed, the process is reversed and the results are returned back to the client.

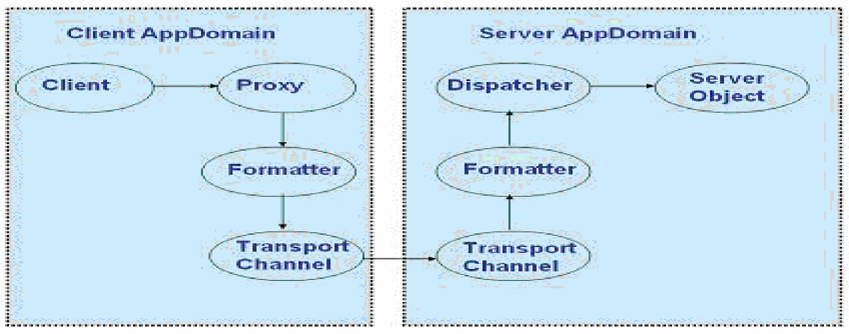


Figure 8 .Net Remoting Architecture

In the Web Services architecture (see Figure 9) the client proxy receives the request from the client, serializes the request into a SOAP which is then forwarded to the remote Web service. The remote Web service receives the SOAP request, executes the method, and sends the results in the form of a SOAP response to the client proxy, which deserializes the message and forwards the actual results to the client.

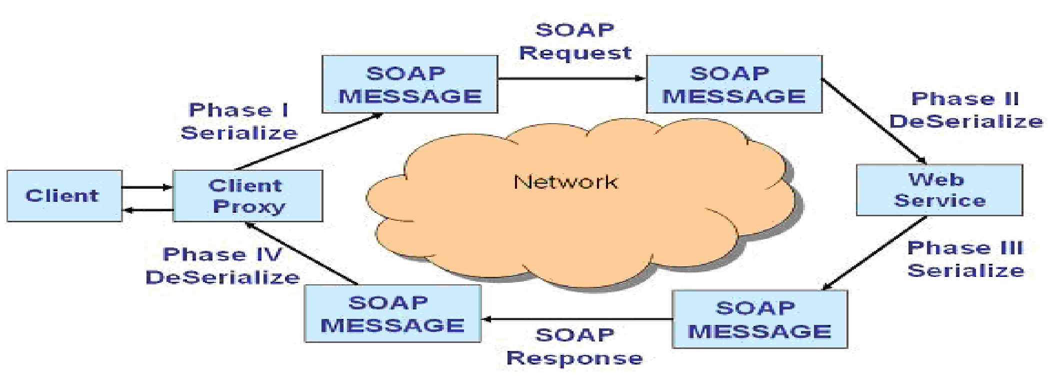


Figure 9 Web Services Architecture

Figure 10, shows a table which compares Web Services and .Net Remoting. Web Services only support the HTTP protocol while .Net Remoting supports the TCP, the HTTP and SMTP protocols. Therefore, considering this property .Net Remoting is better suited than Web Services. On the performance side [11], the Web Services has better performance in comparing the performance of .Net Remoting using the HTTP and TCP channels. Web Services are stateless. Each time a client invokes an ASP.NET Web service, a new object is created to service the request. The object is destroyed after the method call completes. On the other side, .Net Remoting can have stateless objects and no stateless objects. Such the remoting infrastructure allows you to create Singleton (state of a single instance is shared among clients) remote objects and SingleCall (stateless) objects.

Web Services are characterized to be platform independent. The simple programming model based on mapping SOAP message exchanges to individual method invocations makes it possible to use the WSDL (Web Services Definition Language) and XML schema. However, .Net Remoting is a complex model which both sides to communicate need the .Net Infrastructure.

	Web Services	.Net Remoting
Protocol	Can be only accessed over HTTP	Can be accessed over any protocol (including TCP, HTTP, SMTP and so on)

	Web Services	.Net Remoting
Performance	Better performance using the SOAP formatter with either the HTTP or the TCP channel (comparing when used with .Net Remoting)	Better performance using TCP channels because Web Services do not support them
State Management	Web services work in a stateless environment	Provide support for both stateful and stateless environments through Singleton and SingleCall objects
Type	Web services support only the datatypes defined in the XSD type system, limiting the number of objects that can be serialized.	Using binary communication, .NET Remoting can provide support for rich type system
Intereroperability	Web services support interoperability across platforms, and are ideal for heterogeneous environments.	.NET remoting requires the client be built using .NET, enforcing homogenous environment.
Reliability	Highly reliable due to the fact that Web services are always hosted in IIS	Can also take advantage of IIS for fault isolation. If IIS is not used, application needs to provide plumbing for ensuring the reliability of the application.
Extensibility	Provides extensibility by allowing us to intercept the SOAP messages during the serialization and deserialization stages	Very extensible by allowing us to customize the different components of the .NET remoting framework.
Ease of Programming	Easy to create and deploy.	Complex to programme.

Figure 10 Comparing ASP .Net Web Services with .Net Remoting

CHAPTER 3. TRANSFORMING PRISMA DISTRIBUTION MODEL TO C# AND .NET REMOTING

3.1 Introduction

In previous works, the PRISMA distribution model has been defined [1][2][3] . In addition, a UML profile has been defined in order to represent it with the UML notations and concepts. To automatically generate distributed applications code, the MDA proposal is going to be used. The technology chosen to be generated is C# .Net [5] and the .Net Remoting framework [21].

Figure 11, shows the steps that have to be followed to generate PRISMA distributed applications in C# .Net following the MDA proposal. Nevertheless, the steps shown are exactly the same for generating the applications in any technology. Following the MDA the PRISMA distribution model is at a PIM level. To transform the PIM to a PSM, some technology patterns are identified. At a PSM, the model is represented using a logical diagram. Finally, some implementation patterns are used to automatically generate the code of the distributed applications in C# .Net Remoting.

In this chapter, the transformation patterns for automatically generating distributed and mobile applications from the PRISMA distribution model are identified. In the first section, the PSM of the PRISMA distribution model is discussed showing the logical diagram of the model. The technology patterns of transforming the PIM to PSM are done through a table showing the mappings between them. Next, the implementation patterns for implementing distributed applications in C# are identified by using a pattern template. To identify these implementation patterns, the PRISMA distribution model has been implemented. In Appendix B, a bank system example is presented to show how it is implemented using the PRISMA classes.

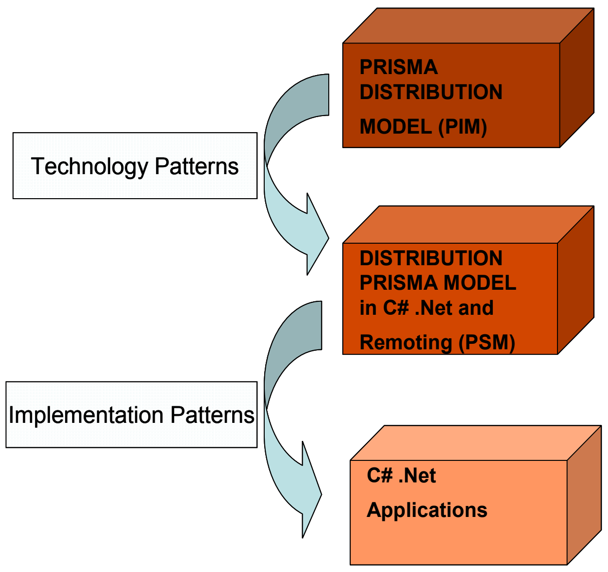

PRISMA DISTRIBUTION MODEL (PIM)
Technology Patterns
DISTRIBUTION PRISMA MODEL in C# .Net and Remoting (PSM)
Implementation Patterns
C# .Net Applications

Figure 11 Using MDA to generate PRISMA distributed applications in C# .Net.

3.2 Platform Specific Model of the PRISMA Distribution Model in C# .Net

The first step for creating a PSM is to choose a target platform. In this work, the target platform is .Net using the C# programming language. As we are going to generate distributed applications, we are going to use .Net Remoting. Our preference was using .Net Remoting to Web Services is due to the fact that web services implement a service-oriented architecture and not a component-based architecture as ours. In addition, another fundamental fact that has influenced in our choice was that web services are stateless however .Net Remoting can manage objects with states.

To implement the PRISMA private services, which the .Net platform does not easily provide, a PRISMA middleware is necessary. The PRISMA middleware (see Figure 12) is an abstract middleware that sits above the .Net platform. As the PRISMA private services, are not directly provided by the .Net platform, the middleware hides the complexity of the implementation of these functionalities. The PRISMA middleware is implemented using the .Net platform, using .Net Remoting and programmed in C#.

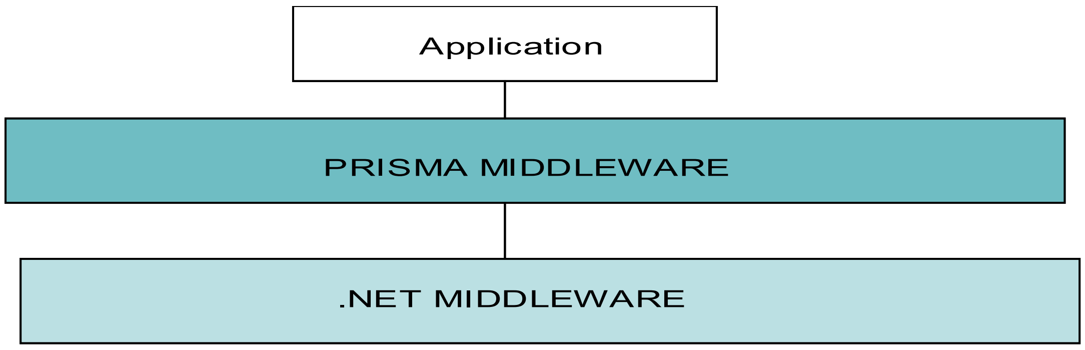


Figure 12 PRISMA Middleware

The abstract middleware contains all the constructs necessary to create and manage PRISMA applications. In addition, the PRISMA middleware is in charge of evolution and reconfiguration of the architectural models. Figure 13, shows the packages that contain the classes and other constructs of the PRISMA middleware. For example to create a specific PRISMA aspect, the class *AspectBase* of the middleware has to be extended.

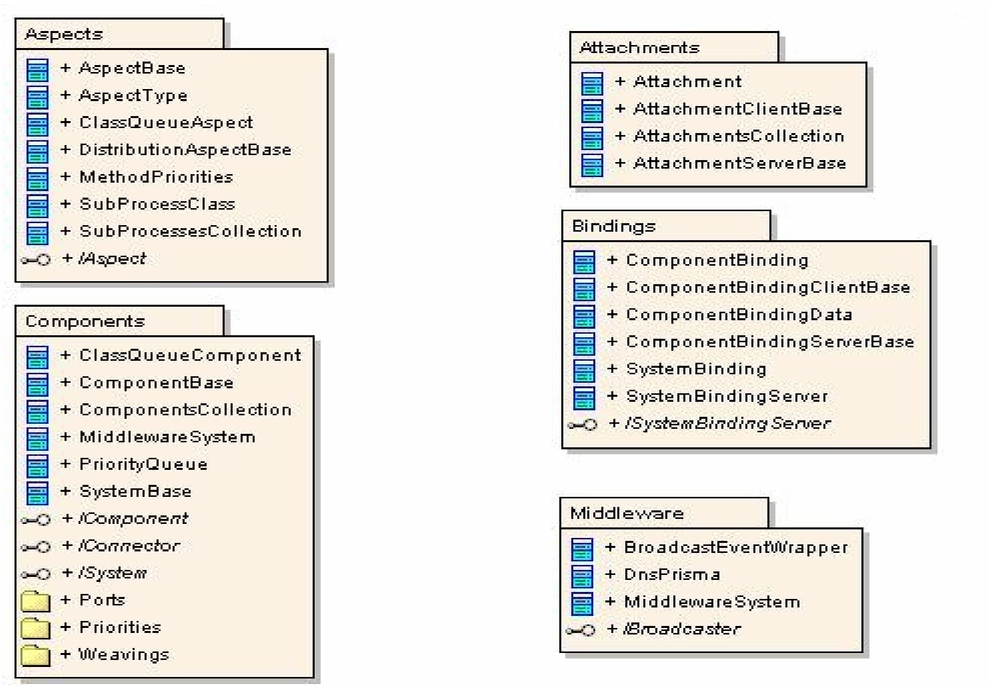


Figure 13 Set of Constructs of the PRISMA middleware

As the PRISMA framework can be distributed, each host where the PRISMA framework is running should have the PRISMA middleware executing. The PRISMA architectural elements use the services of their local middlewares and can also call remote middlewares through their local one. Therefore, the PRISMA middlewares' of an architectural model use .Net Remoting to communicate among them. The middleware systems uses the namespace *System.Runtime.Remoting* to use the classes and interfaces that it provides to enable distributed communication and configuration. There is only a middleware instance running at a time, therefore the calls among the middlewares are Server Activator Object (SAO) and are activated by *Singleton*. Singleton types only can have one instance at a time. Thus, all clients are served by the same instance.

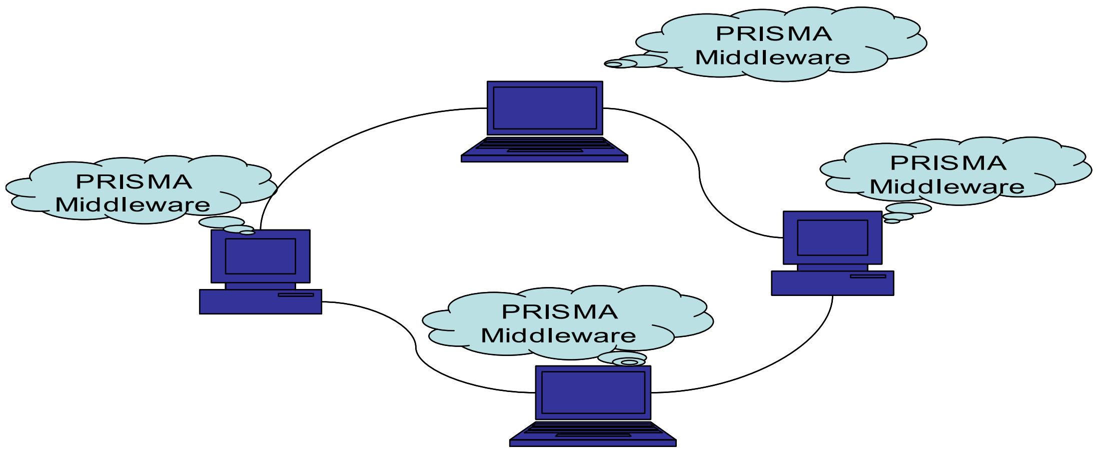


Figure 14 PRISMA Middleware running on each machine to enable a distribution model.

The distribution PRISMA model (PIM) has been mapped to C# .Net concepts. Figure 15, shows the mapping of the distribution PRISMA model to C#. These C# concepts form part of the PRISMA middleware. Thus, the middleware contains the classes and services to enable the creation and management of PRISMA plus the proper services that allow distribution, mobility, evolution and reconfiguration.

PRISMA DISTRIBUTION MODEL ELEMENT (PIM)	**PRISMA DISTRIBUTION MODEL ELEMENTS IN C# (PSM)**
PRISMA METAMODEL	MIDDLEWARE(A set of classes)
Distribution Aspect	Class
LOC data type	Structure
Move	A set of methods that call to the component and middleware
Attachments	2 sealed classes which aggregate 2 classes: a client class and server class. class(MarshalByRef)
Binding Links	2 classes: A class for the binding on the system side and another for the binding of the architectural element the system contains. The class of the architectural element side that the system contains is composed of two parts: a client side and a server side (MarshalByRef).

Figure 15 Mappings between the PRISMA distribution model and C#.

In the following, the Distribution Specific model is going to be presented in detail. The Loc data type and distribution aspect specific model are explained. In addition, Attachments Specific Model, the Bindings Specific Model and the Mobility Specific Model in .Net C# are going to be presented showing how the Middleware participates in each one of them. The specific models are explained using logical views.

3.2.1 loc Data type Specific Model

The PRISMA abstract data type “loc” specifies if a “location” is valid or not. In .Net the data type is implemented as a structure (see Figure 16). As .Net Remoting uses the tcp and http protocol, the *loc* structure checks if the location value is either tcp or http. In addition, the any attribute of type “loc”, should have a value. Therefore, the loc checks that the attribute does not have a null value.

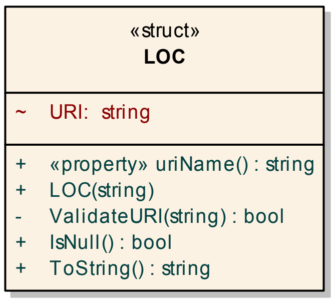


Figure 16 The loc data type.

3.2.2 Distribution Aspect Specific Model

The distribution aspect specifies the features and strategies that manage the dynamic location of the instances of the architectural elements of a software architecture. The distribution extends the *AspectBase* class. As the distribution aspect has the same features of any aspect plus its own properties such as the *location*, the *DistributionAspectBase* inherits from *AspectBase* (see Figure 17).

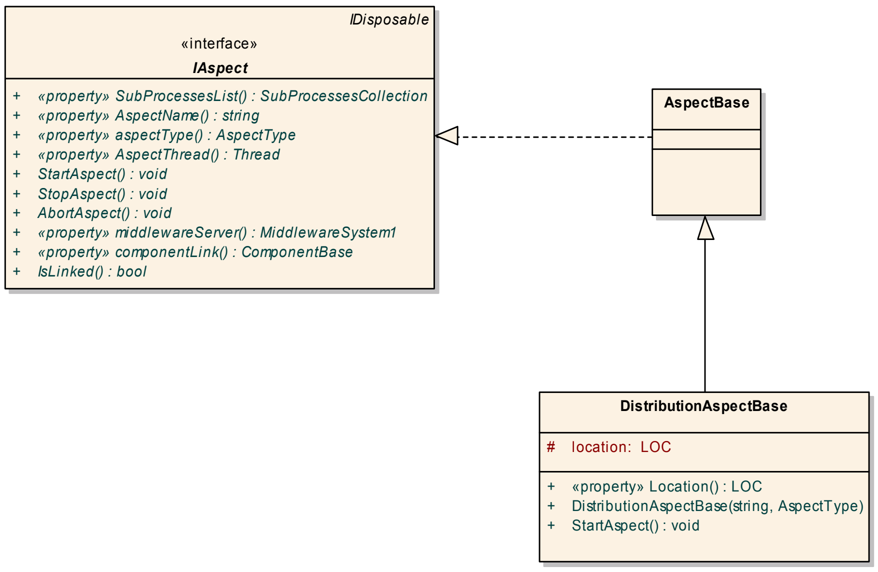

IDisposable
«interface»
IAspect
+ «property» SubProcessesList() : SubProcessesCollection
+ «property» AspectName() : string
+ «property» aspectType() : AspectType
+ «property» AspectThread() : Thread
+ StartAspect() : void
+ StopAspect() : void
+ AbortAspect() : void
+ «property» middlewareServer() : MiddlewareSystem1
+ «property» componentLink() : ComponentBase
+ IsLinked() : bool
AspectBase
DistributionAspectBase
# location: LOC
+ «property» Location() : LOC
+ DistributionAspectBase(string, AspectType)
+ StartAspect() : void

Figure 17 The Distribution Aspect logical view.

As a distribution aspect can specify the mobility of an instance, another class has been defined for this case. Therefore, another class called *MobileDistributionAspect* inherits from *DistributionAspectBase.* The *MobileDistributionAspect* maps a distribution aspect with a move service (see Figure 18).

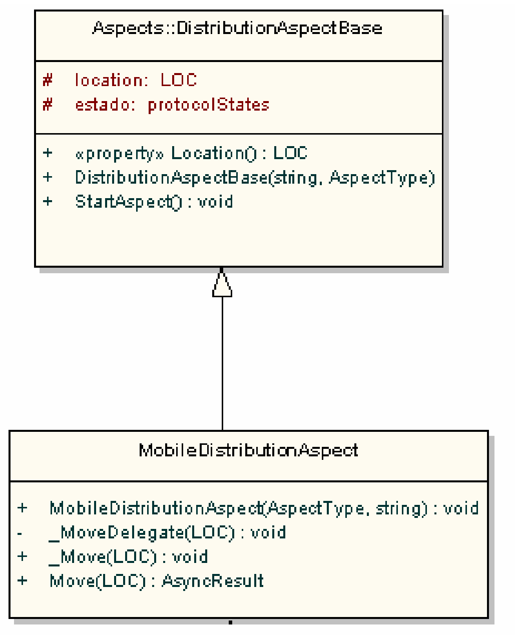

Aspects::DistributionAspectBase
# location: LOC
# estado: protocolStates
+ «property» Location() : LOC
+ DistributionAspectBase(string, AspectType)
+ StartAspect() : void
MobileDistributionAspect
+ MobileDistributionAspect(AspectType, string) : void
- _MoveDelegate(LOC) : void
+ _Move(LOC) : void
+ Move(LOC) : AsyncResult

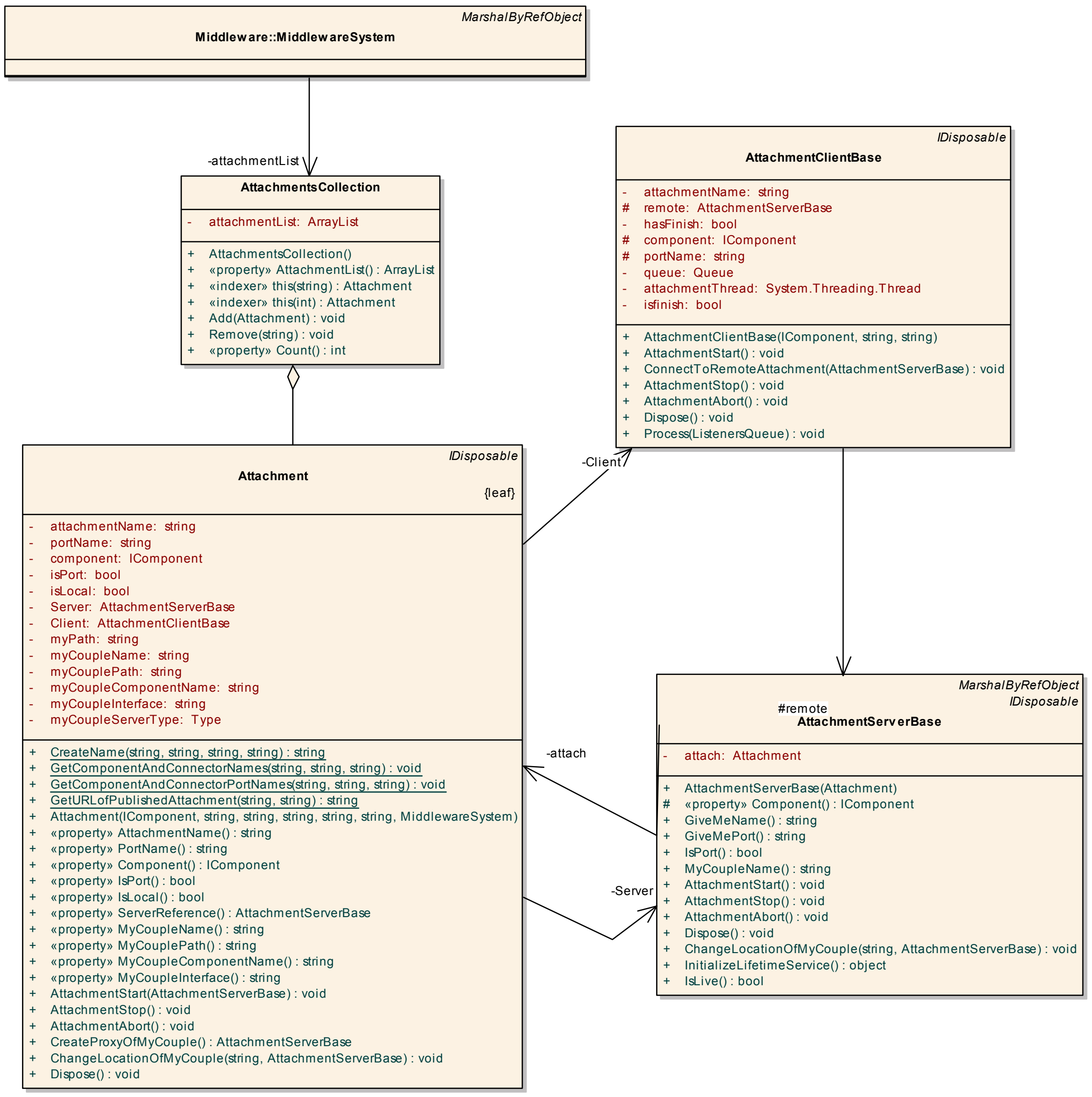
Figure 18 The Distribution Aspect with a move service.

3.2.3 Attachments Specific Model

An attachment in the PRISMA distribution model connects two distributed architectural element instances which need to communicate. Thus, an attachment can be seen as the communication channel between two instances of the architectural model. An attachment is defined by connecting a port (A port is the point where a component receives and offers its services) of a component with a role of a connector. If the architectural elements are distributed then their locations should be indicated.

Four classes are necessary to exist for implementing PRISMA attachments (see Figure 19). The *AttachmentsCollection* class is necessary so that the middleware keeps the list of Attachments defined on that machine (middleware). The *Attachment* class represents part of the Attachment of an architectural element. Thus a PRISMA attachment in C# is mapped into two Attachment classes. Each attachment class is to represent an attachment side of an architectural element. Therefore, a PRISMA attachment can be mapped into two distributed classes depending if the architectural elements are distributed. For example, a specification of a PRISMA attachment “Comp1(Port1,location) ←→ Conn1(Port2,location)” is implemented by an *Attachment* class at the Comp1 architectural element and another at the side of the Conn1 architectural element. If the architectural elements are distributed then each middleware has a reference of the Attachment side of the architectural element which resides on the same machine.

Each Attachment side has a reference of the other side of the Attachment. However, each Attachment side creates an instance of an *AttachmentClientBase* and an instance of an *AttachmentServerBase* on its side. The *AttachmentClientBase* is a thread that listens at a certain port of the attached architectural element and redirects the methods to an AttachmentServerBase of the other side of the attachment. The *AttachmentServerBase* is the part of the attachment that is published by remoting through the *MarshalByRef* class. Thus, the *AttachmentServerBase* is the part of the attachment which redirects the method to a port of an architectural element. In addition, in many cases it acts as an intermediary between the exterior and the *AttachmentClientBase* instance thus as previously commented it is published by Remoting.



MarshalByRefObject
Middleware::MiddlewareSystem
-attachmentList
AttachmentsCollection
- attachmentList: ArrayList
+ AttachmentsCollection()
+ «property» AttachmentList() : ArrayList
+ «indexer» this(string) : Attachment
+ «indexer» this(int) : Attachment
+ Add(Attachment) : void
+ Remove(string) : void
+ «property» Count() : int
IDisposable
AttachmentClientBase
- attachmentName: string
# remote: AttachmentServerBase
- hasFinish: bool
# component: IComponent
# portName: string
- queue: Queue
- attachmentThread: System.Threading.Thread
- isfinish: bool
+ AttachmentClientBase(IComponent, string, string)
+ AttachmentStart() : void
+ ConnectToRemoteAttachment(AttachmentServerBase) : void
+ AttachmentStop() : void
+ AttachmentAbort() : void
+ Dispose() : void
+ Process(ListenersQueue) : void
IDisposable
Attachment
{leaf}
- attachmentName: string
- portName: string
- component: IComponent
- isPort: bool
- isLocal: bool
- Server: AttachmentServerBase
- Client: AttachmentClientBase
- myPath: string
- myCoupleName: string
- myCouplePath: string
- myCoupleComponentName: string
- myCoupleInterface: string
- myCoupleServerType: Type
+ CreateName(string, string, string, string) : string
+ GetComponentAndConnectorNames(string, string, string) : void
+ GetComponentAndConnectorPortNames(string, string, string) : void
+ GetURLofPublishedAttachment(string, string) : string
+ Attachment(IComponent, string, string, string, string, string, MiddlewareSystem)
+ «property» AttachmentName() : string
+ «property» PortName() : string
+ «property» Component() : IComponent
+ «property» IsPort() : bool
+ «property» IsLocal() : bool
+ «property» ServerReference() : AttachmentServerBase
+ «property» MyCoupleName() : string
+ «property» MyCouplePath() : string
+ «property» MyCoupleComponentName() : string
+ «property» MyCoupleInterface() : string
+ AttachmentStart(AttachmentServerBase) : void
+ AttachmentStop() : void
+ AttachmentAbort() : void
+ CreateProxyOfMyCouple() : AttachmentServerBase
+ ChangeLocationOfMyCouple(string, AttachmentServerBase) : void
+ Dispose() : void
-Client
-attach
-Server
MarshalByRefObject
IDisposable
#remote
AttachmentServerBase
- attach: Attachment
+ AttachmentServerBase(Attachment)
# «property» Component() : IComponent
+ GiveMeName() : string
+ GiveMePort() : string
+ IsPort() : bool
+ MyCoupleName() : string
+ AttachmentStart() : void
+ AttachmentStop() : void
+ AttachmentAbort() : void
+ Dispose() : void
+ ChangeLocationOfMyCouple(string, AttachmentServerBase) : void
+ InitializeLifetimeService() : object
+ IsLive() : bool

Figure 19 Logical view of PRISMA Attachments

Figure 20, shows an interaction diagram of the commencement of the execution of an attachment. The figure assumes that the two Attachment instances are distributed therefore the instance RemoteMiddleware exists to represent the distributed communication between it and the local Middleware (MiddlewareSystem). When a middleware is instantiated, the middleware directly instantiates the *AttachmentsCollection* class. The Middleware has a method that creates Attachments *createAttachment*. The middleware checks if the two sides of the attachment are local or not. If the two sides of the attachment is distributed it invokes the *createAttachment* of the middleware where the other architectural element resides (RemoteMiddleware) to instantiate the attachment part on its side. Then the middleware instantiates the class *Attachment*, for the attachment part on its side, in which the Attachment instantiates *AttachmentClientBase* and *AttachmentServerBase*.

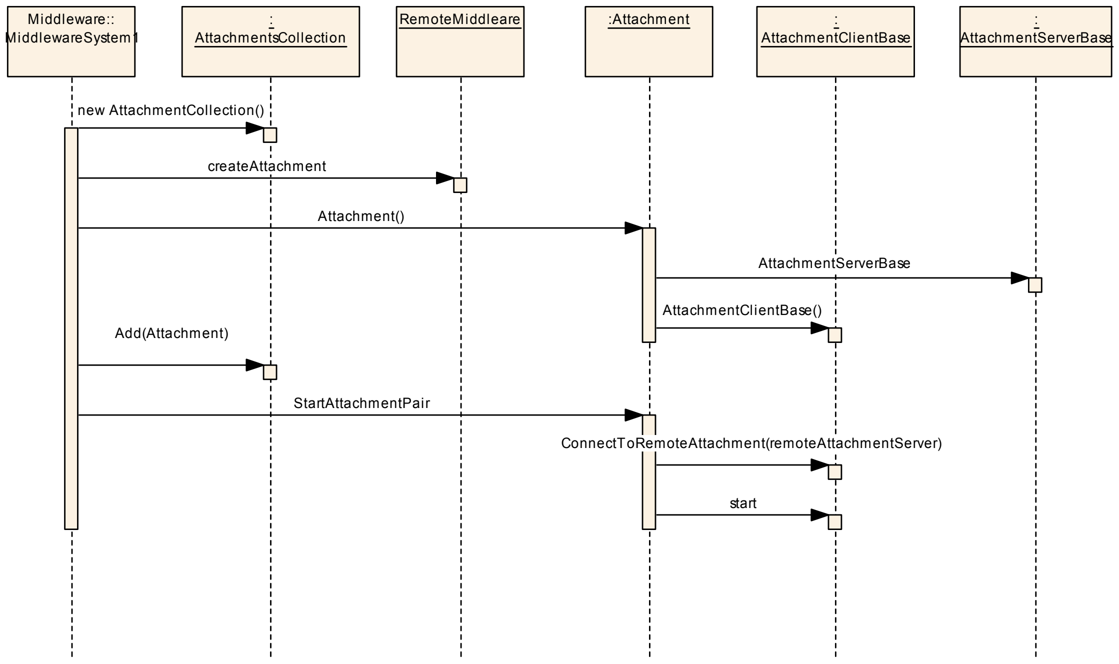


Figure 20 An interaction diagram of the Attachments

After, the middleware invokes the *AttachmentStart* of the *AttachmentServerBase* instance of the other middleware to redirect it to the *Attachment* of its side. The middleware calls the *AttachmentServerBase* of the other side because it is published by remoting and is accessible. Then the middleware invokes the *startAttachment* of the Attachment instance. The *Attachment* then calls the start of the *AttachmentClientBase* instance so that it registers to a port of the architectural element and creates a thread to listen at that port.

3.2.4 Bindings Specific Model in C#

A binding is the connection between a PRISMA *system* (an architectural element which is composed of other architectural elements) and the architectural element it contains. Thus, a binding is the communication channel between an architectural element of a *system* and the *system* it belongs to. In PRISMA, the system canbe independent of the architectural elements it contains therefore, a system and its architectural elements can be distributed. A binding in PRISMA is defined by connecting a system port with a component port. If they are distributed then the locations of the system and component has to be given.

When a PRISMA system is created a *SystemBinding* is instantiated. The *SystemBinding* class represents the part of the binding on the system side (see Figure 21). That is, the system side has its own binding part which is the *SystemBinding*. The *ComponentBinding* is the class which represents the binding part of the architectural element which is contained in the system. If the system architectural element and the architectural element it contains are distributed the *SystemBindingServer* is instantiated. The *ComponentBinding* creates an instance of the *ComponentBindingClientBase* and the *ComponentBindingServerBase*. For example, a specification of a PRISMA binding "System(Port1,location) ←→ Comp1(Port2,location)" is implemented by an *SystemBinding* class at the System architectural element and another at the side of the Comp1 architectural element which is implemented by the *ComponentBinding*.

The *ComponentBindingClientBase* is a thread that listens at a certain port of the attached architectural element and redirects the methods to a *SystemBindingServer* of the other side of the binding if the system and the architectural element it contains are distributed. The *ComponentBindingServerBase* is the part of the binding of a contained architectural element that is published by remoting through the MarshalByRef class. Thus the *ComponentBindingServerBase* is the part of the binding which redirects the method to a port of an architectural element. In addition, in many cases it acts as an intermediary between the exterior and the *ComponentBindingClientBase* instance thus as previously commented it is published by remoting. The structure *ComponentBindingData* is used by the *SystemBinding* to store the necessary information of a *ComponentBinding* for example the ports that a binding is attached to.

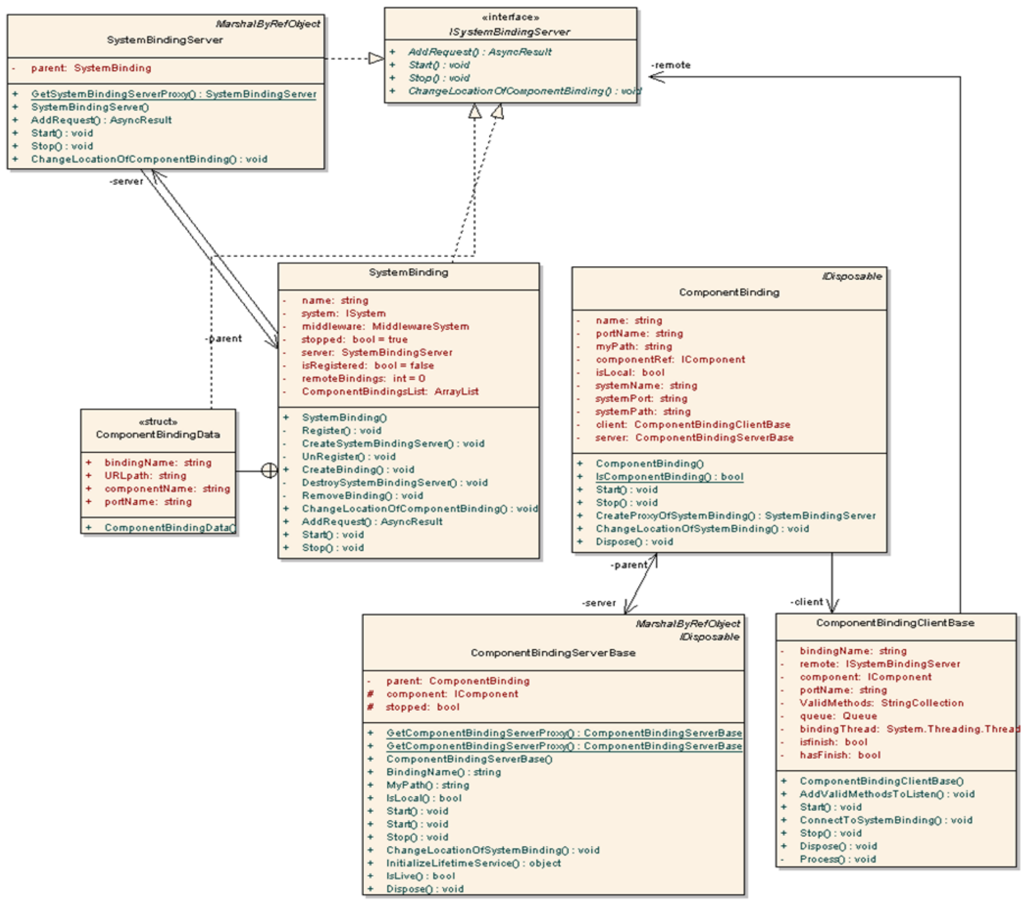

MarshalByRefObject
SystemBindingServer
- parent: SystemBinding
+ GetSystemBindingServerProxy() : SystemBindingServer
+ SystemBindingServer()
+ AddRequest() : AsyncResult
+ Start() : void
+ Stop() : void
+ ChangeLocationOfComponentBinding() : void
«interface»
ISystemBindingServer
+ AddRequest() : AsyncResult
+ Start() : void
+ Stop() : void
+ ChangeLocationOfComponentBinding() : void
-remote
-server
-parent
SystemBinding
- name: string
- system: ISystem
- middleware: MiddlewareSystem
- stopped: bool = true
- server: SystemBindingServer
- isRegistered: bool = false
- remoteBindings: int = 0
- ComponentBindingsList: ArrayList
+ SystemBinding()
- Register() : void
- CreateSystemBindingServer() : void
- UnRegister() : void
+ CreateBinding() : void
- DestroySystemBindingServer() : void
- RemoveBinding() : void
+ ChangeLocationOfComponentBinding() : void
+ AddRequest() : AsyncResult
+ Start() : void
+ Stop() : void
«struct»
ComponentBindingData
+ bindingName: string
+ URLpath: string
+ componentName: string
+ portName: string
+ ComponentBindingData()
IDisposable
ComponentBinding
- name: string
- portName: string
- myPath: string
- componentRef: IComponent
- isLocal: bool
- systemName: string
- systemPort: string
- systemPath: string
- client: ComponentBindingClientBase
- server: ComponentBindingServerBase
+ ComponentBinding()
+ IsComponentBinding() : bool
+ Start() : void
+ Stop() : void
+ CreateProxyOfSystemBinding() : SystemBindingServer
+ ChangeLocationOfSystemBinding() : void
+ Dispose() : void
-parent
-server
-client
MarshalByRefObject
IDisposable
ComponentBindingServerBase
- parent: ComponentBinding
# component: IComponent
# stopped: bool
+ GetComponentBindingServerProxy() : ComponentBindingServerBase
+ GetComponentBindingServerProxy() : ComponentBindingServerBase
+ ComponentBindingServerBase()
+ BindingName() : string
+ MyPath() : string
+ IsLocal() : bool
+ Start() : void
+ Start() : void
+ Stop() : void
+ ChangeLocationOfSystemBinding() : void
+ InitializeLifetimeService() : object
+ IsLive() : bool
+ Dispose() : void
ComponentBindingClientBase
- bindingName: string
- remote: ISystemBindingServer
- component: IComponent
- portName: string
- ValidMethods: StringCollection
- queue: Queue
- bindingThread: System.Threading.Thread
- isfinish: bool
- hasFinish: bool
+ ComponentBindingClientBase()
+ AddValidMethodsToListen() : void
+ Start() : void
+ ConnectToSystemBinding() : void
+ Stop() : void
+ Dispose() : void
- Process() : void

Figure 21 Logical View of a Binding implemented in C#

Figure 22, shows an interaction diagram of how the bindings start to work. When a *SystemBase* is instantiated it instantiates a *SystemBinding* to add the bindings associated to it. The *SystemBinding* instantiates a *SystemBindingServer* if the two binding parts are distributed, if they are not distributed it is not necessary to create an instance of *SystemBindingServer*. The *SystemBinding* asks the middleware to create a *ComponentBinding*. The local middleware (*MiddlewareSystem1*) checks if the component resides on the same machine as the middleware or not. If the component is distributed the middleware calls the *createComponentBinding* of the other middleware where the component resides. The *ComponentBinding* creates an instance of *ComponentBindingClientBase* and *ComponentBindingServerBase*. If the component is local the middleware returns a reference of the *ComponentBindingServerBase* to the *SystemBinding* if not it returns a proxy to the *ComponentBindingServerBase* of the other side.

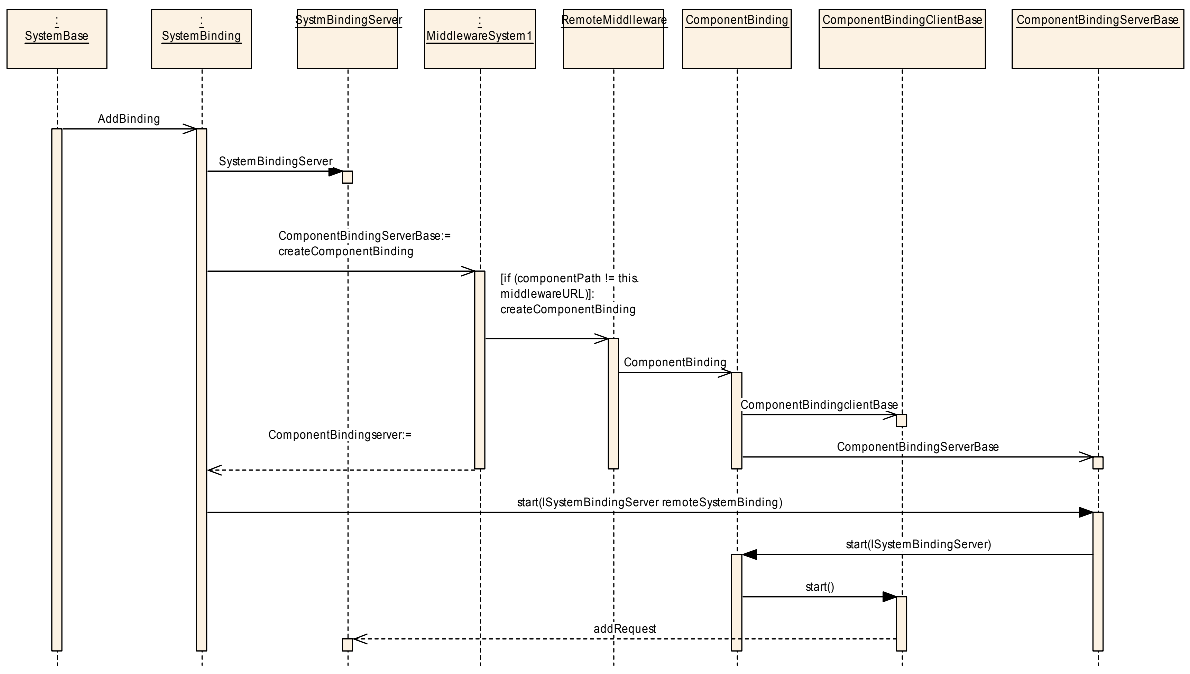

: SystemBase
: SystemBinding
SystmBindingServer
: MiddlewareSystem1
RemoteMiddlleware
ComponentBinding
ComponentBindingClientBase
ComponentBindingServerBase
AddBinding
SystemBindingServer
ComponentBindingServerBase:= createComponentBinding
[if (componentPath != this. middlewareURL)]: createComponentBinding
ComponentBinding
ComponentBindingclientBase
ComponentBindingserver:=
ComponentBindingServerBase
start(ISystemBindingServer remoteSystemBinding)
start(ISystemBindingServer)
start()
addRequest

Figure 22 An interaction diagram of the Bindings

When the middleware returns a reference or a proxy to the *SystemBinding*, the *SystemBinding* stores the information of the *ComponentBinding* in a structure. If the *ComponentBinding* is distributed the *SystemBinding* calls *start(ISystemBindingServer)* to the *ComponentBindingServerBase*. This is done due to the fact that the *ComponentBindingServerBase* is published through Remoting and can be accessible. Then, *ComponentBindingServerBase* redirects this message to the *ComponentBinding*. The *ComponentBinding* then calls the *ConnectToSystemBinding* of the *ComponentBindingClientBase* to give it the reference of the *SystemBindingServer*. After, the *ComponentBinding* calls the start of the *ComponentBindingClientBase* to register it to a component port and creates its thread to start listen. When a *ComponentBindingClientBase* detects a method in a port it processes it and redirects it to the *SystemBindingServer* through the method *AddRequest* which then is redirected to the *SystemBinding*.

3.2.5 Mobility Specific Model in C#

The Mobility Specific Model of PRISMA in C# .Net is characterized to be a Strong Mobility (previously explained in section 2.4). Strong Mobility has been chosen as the form of mobility provided by PRISMA to the fact that in our model we are interested to transfer the state of the mobile objects. In addition our objective is to support replication and migration. The migration supported by PRISMA is both proactive and reactive. It is proactive and reactive because we enable that the same object or other objects of the PRISMA architectural model decide on the time and destination of the migrant.

The principal participants of the Mobility Model are the *MobileDistributionAspect*, the component thread and the two middlewares: the local middleware and the middleware where the component is going to be moved (see Figure 23). Each class that its instances have to be moved, have to be marked with [Serializable] attribute. In our model, all components and aspects have to be marked with [Serializable] attribute.

A move method of the *MobileDistributionAspect* calls the Component thread to indicate it that it has to process a move. When the distribution aspect notifies the component thread of the mobility the Component thread stops processing from its queue. However, services can be queued in the Component queue because the queue is Serializable. Then the component notifies its aspects to stop their threads when they finish processing their queues. When the whole component stops it notifies the middleware so that the middleware starts the mobility process. Then the middleware looks for all the attachments and bindings associated to the mobile component. The middleware performs this operation by checking all the ports that have a registered attachment. The middleware invokes the Stop method of the Attachment to stop them executing and unregisters the attachment part local to it.

Next, the local middleware calls the middleware where the component is going to be moved. The other middleware creates the component and its associated attachments and registers them on its side. Then for each ServerAttachment the new reference or proxy is sent to the ClientAttachment on the other side.

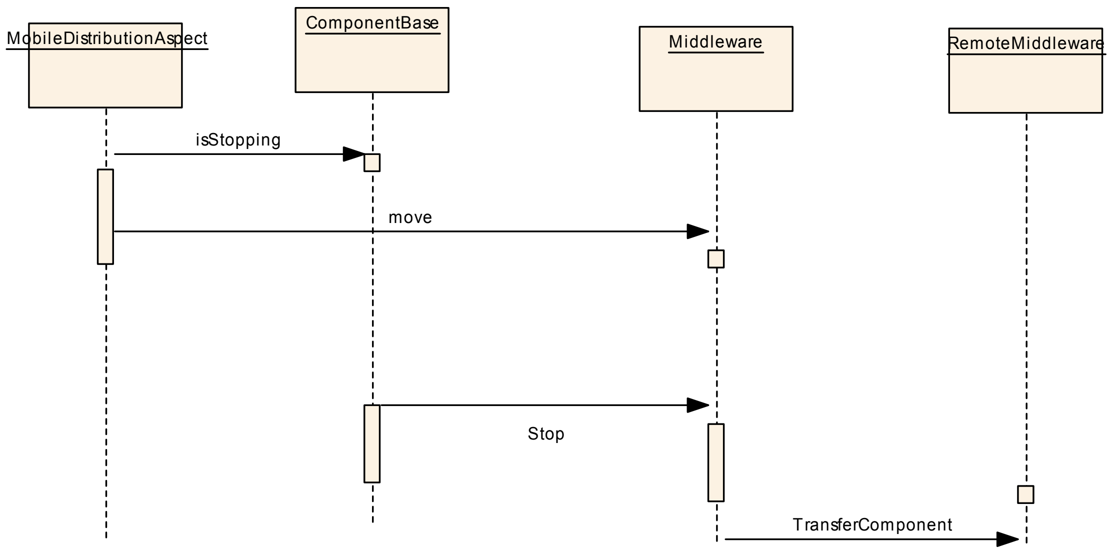


Figure 23 An Interaction Diagram of the Mobility Process

3.3 Implementation Patterns

This section identifies the implementation patterns of the PRISMA distribution model. These patterns describe how certain elements of the PRISMA applications are automatically generated using some implemented C# classes that form the PRISMA middleware.

The patterns template used in this section, follows the pattern template in [20]. The following sections are part of the template:

- **Pattern Name:** A name is given to describe the problem solved.
- **Context:** Where and when the pattern is applied.
- **Problem:** What problems does the pattern solve?
- **Forces:** What makes it necessary to find for a solution to the problem
- **Solution:** How is the problem solved?
- **Structure:** A graphical representation of the solution of the problem.
- **Participants:** The classes that participate in the solution.
- **Consequences:** How does the pattern support its objectives?
- **Example:** An example is used to explain to show how the pattern is used.
- **Implementation:** This section shows some explanations and comments on how the pattern has been implemented.
- **Related Patterns:** What patterns are related to this pattern?

3.3.1 loc Data Type.

Context.

A data type called loc of the PRISMA meta-model has to be implemented in order to automatically generate properties with data type loc. This data type checks if the locations assigned to the PRISMA elements have a correct position in the PRISMA application system. This data type has to be implemented in C#.

Problem.

How is the loc data type implemented in C#? How can a property have a data type of type loc?

Forces.

- Maintenance effort: very low maintenance costs are desirable. The properties with a data type loc have to be automatically generated.
- Reusability: The solution proposed should be reusable for all properties with data type loc.
- Flexibility: the solution proposed should be adaptable to work with all data types PRISMA and C#

Solution.

A C# structure called loc is defined. Each class that has a property with data type loc has to be associated with loc structure. The only PRISMA class that can have a loc data type is the distribution aspect.

Structure.

Figure 24 shows a generic class diagram which uses the solution proposed. The structure loc is responsible of giving a data type of type loc to any property.

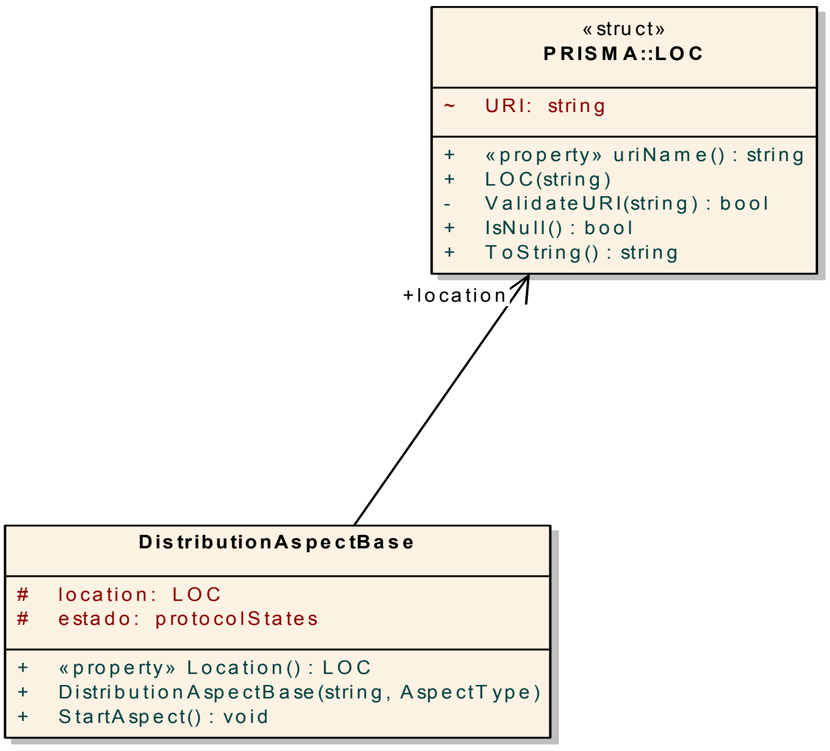


Figure 24 Generic use of loc

Participants.

In Figure 24:

- *A* is a property which instantiates the structure loc in order to assign its data type as loc.
- *LOC* is the structure that checks if the property is valid to be a correct location in respect to the PRISMA application system.

Consequences.

- Facilitates automatic generation of properties of data type loc. To make a property of data type loc all that is required is to make an instantiation of the structure LOC.
- Good maintainability. The property is checked to be a valid location.
- Reusability. LOC is directly reusable by any property that has to be of data type loc.

Example.

Figure 25, shows how a distribution aspect has an association with the *LOC* structure to have a location property with a loc data type.

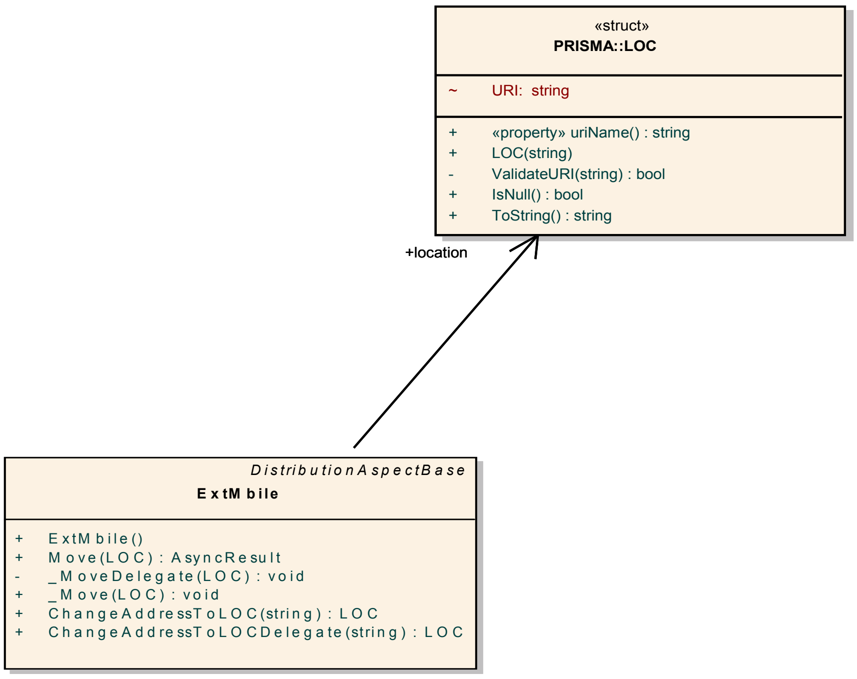

«struct»
PRISMA::LOC
~ URI: string
+ «property» uriName() : string
+ LOC(string)
- ValidateURI(string) : bool
+ IsNull() : bool
+ ToString() : string
+location
DistributionAspectBase
ExtMbile
+ ExtMbile()
+ Move(LOC) : AsyncResult
- _MoveDelegate(LOC) : void
+ _Move(LOC) : void
+ ChangeAddressToLOC(string) : LOC
+ ChangeAddressToLOCDelegate(string) : LOC

Figure 25 A distribution aspect is associated with loc structure

Implementation.

- *public* **uriName**():string

This is a property associated to a string which represents the location. It sets the location after checking if it has a valid value. It also returns the value if it is requested. The code is as following:

```
            string URI;
            public string uriName {
                  get { return URI; }
                  set { URI = (ValidateURI(value) ? value :
"");  }

            }
```

- *public* **LOC** (string location)

This is the constructor method of LOC. It has as a parameter a location. It validates the parameter through the method *`ValidateURI(location)`*, `only` if the location is valid it assigns it to the URI. Its code is as following:

```
         public LOC(string location){
                  URI = null;
                  if (ValidateURI(location)) {
                        URI = location;
                  }
      }
```

- *private* **ValidateURI** (string location):bool

This method checks if the location is valid. The correctness is checked if the location is an http or a tcp. This is done by checking if the string starts with an *http://* and a *tcp://.* Its code is as following:

```
         private bool ValidateURI(string location) {
                  if (!(location.StartsWith("http://")) &&

                 !(location.StartsWith("tcp://"))){
                        throw new
NoValidLOCException(location);
                  } else{
                        return true;
                  }
      }
```

- *public* **IsNull**():bool

This method checks if a value is assigned to location. Its code is as following:

```
public bool IsNull(){
        return (URI == null);

}
```

- *public* **ToString**():string

Related Patterns.

No related Patterns

3.3.2 Distribution Aspect.

Context.

A PRISMA distribution aspect contains the distribution properties and needs to have a variable called location. A distribution aspect has to be implemented to automatically generate the code of a PRISMA distribution aspect. This distribution aspect has to be implemented in C#.

Problem.

How is the distribution aspect implemented in C#? How can a distribution aspect be automatically generated?

Forces.

- Maintenance effort: very low maintenance costs are desirable. The impact of modifying a distribution aspect specification using PRISMA languages and UML notation should be minimum.
- Reusability: The solution proposed should be reusable for all distribution aspects.
- Flexibility: the solution proposed should be adaptable to work with all distribution aspects PRISMA and C#

Solution.

A C# class that inherits from *AspectBase* is defined. Each distribution aspect implemented in C# should inherit from this class.

Structure.

Figure 26, shows a generic class diagram which uses the solution proposed. The *DistributionAspectBase* inherits from AspectBase to specify the proper properties of PRISMA distribution aspects. *DistributionAspectBase* is in charge of generating all PRISMA distribution aspects in C#. The PRISMA distribution aspects have to inherit from *DistributionAspectBase.*

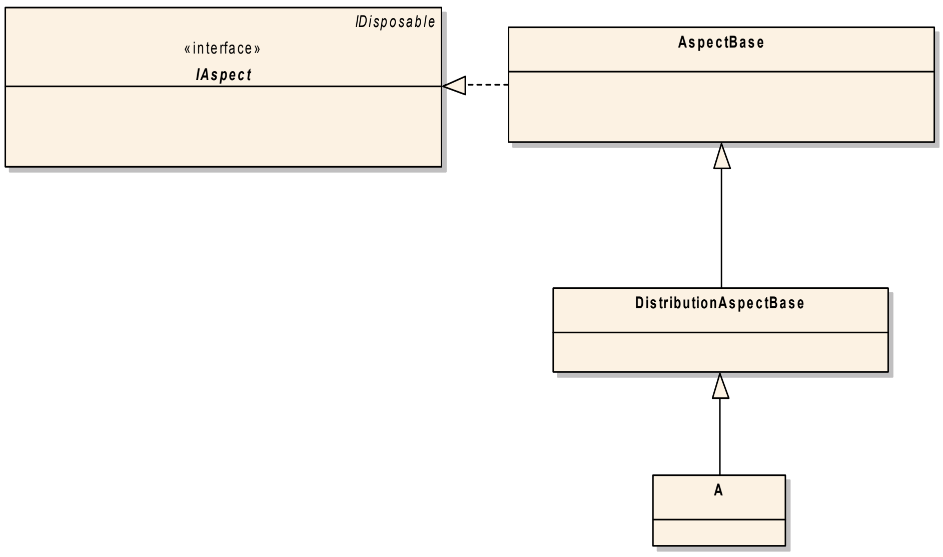


Figure 26 A DistributionAspectBase is implemented that is specialized from AspectBase

Participants.

In Figure 26:

- *A* is a distribution aspect which inherits from *DistributionAspectBase* to become a distribution aspect.
- *DistributionAspectBase* inherits from *AspectBase* to include its own distribution properties that all distribution aspects should specialize to automatically generate the code.
- *AspectBase* is the class that has the properties of a PRISMA aspect.
- *IAspect* is the interface which implements the aspects.

Consequences.

- Facilitates automatic generation of distribution aspects. To make a PRISMA distribution aspect it has to inherit from *DistributionAspectBase*.
- Good maintainability.
- Reusability. Any distribution aspect to be generated has to inherit from *DistributionAspectBase*.

Example.

Figure 27, shows how a distribution aspect called *ExtMbile* has to inherit from *DistributionAspectBase* to be generated.

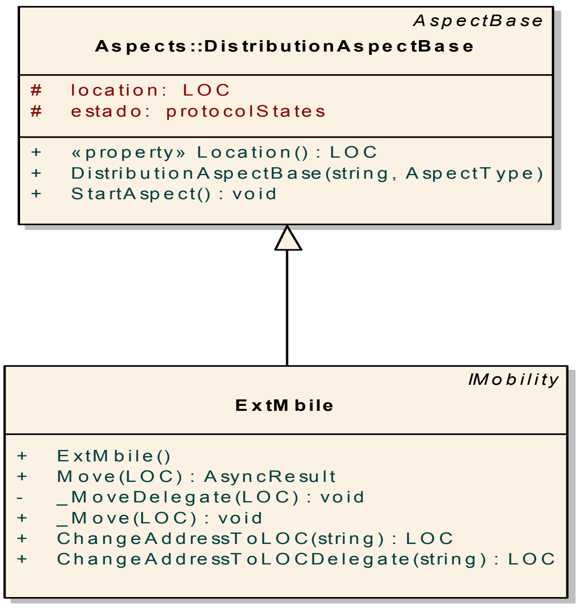

AspectBase
Aspects::DistributionAspectBase
# location: LOC
# estado: protocolStates
+ «property» Location() : LOC
+ DistributionAspectBase(string, AspectType)
+ StartAspect() : void
IMobility
ExtMbile
+ ExtMbile()
+ Move(LOC) : AsyncResult
- _MoveDelegate(LOC) : void
+ _Move(LOC) : void
+ ChangeAddressToLOC(string) : LOC
+ ChangeAddressToLOCDelegate(string) : LOC

Figure 27 A *ExtMbile* distribution aspect has to inherit from the *DistributionAspectBase*

Implementation.

- *public* **Location**():LOC

This is a property to enable to consult the location. It also returns the value if it is requested. The code is as following:

```
protected LOC location;
   public LOC Location {
          get { return location; }
   }
```

• *public* ***DistributionAspectBase****(string name, AspectType type) : base(name , type)*

This is the constructor of DistributionAspectBase class which has to use the constructor of the AspectBase because it is his parent. It has as arguments the name of the distribution aspect and the type to indicate it is a distribution aspect. The code is as following:

```
            public DistributionAspectBase(string name,
AspectType
                type) : base(name , type) {
            }
```

• *public override* ***startAspect****()* :void

This method overrides the *startAspect* method of *AspectBase*. This method is used each time the distribution aspect is loaded for assigning to the *location* property the value of the middleware URL.

```
            public override void StartAspect() {
                  if (

this.aspectType.Equals(AspectType.Distribution))
                        location = new

LOC(link.MiddlewareServer.MiddlewareURL);
                           base.StartAspect();
            }
      }
```

Related Patterns.

No related Patterns

3.3.3 Distribution Aspect with Service move().

Context.

A PRISMA distribution aspect contains the distribution properties and needs to have a variable called location and a move service. The move service enables the mobility of the architectural elements. A distribution aspect has to be implemented to automatically generate the code of a PRISMA distribution aspect with the location and mobility. This distribution aspect has to be implemented in C#.

Problem.

How is the distribution aspect with the move service implemented in C#? How can a distribution aspect with the move servce be automatically generated?

Forces.

- Maintenance effort: very low maintenance costs are desirable. The impact of modifying a distribution aspect specification using PRISMA languages and UML notation should be minimum.
- Reusability: The solution proposed should be reusable for all distribution aspects.
- Flexibility: the solution proposed should be adaptable to work with all distribution aspects PRISMA and C#

Solution.

A C# class that inherits from MobileDistribution*Aspect* is defined. Each distribution aspect with the move service should inherit from this class to be implemented in C#.

Structure.

Figure 28, shows a generic class diagram which uses the solution proposed. The *MobileDistributionAspect* inherits from *DistributionAspectBase* to specify the proper properties of PRISMA distribution aspects including the mobility. *MobileDistributionAspect* is in charge of generating all PRISMA distribution aspects in C# with mobility capabilities. The PRISMA distribution aspects have to inherit from *MobileDistributionAspect.*

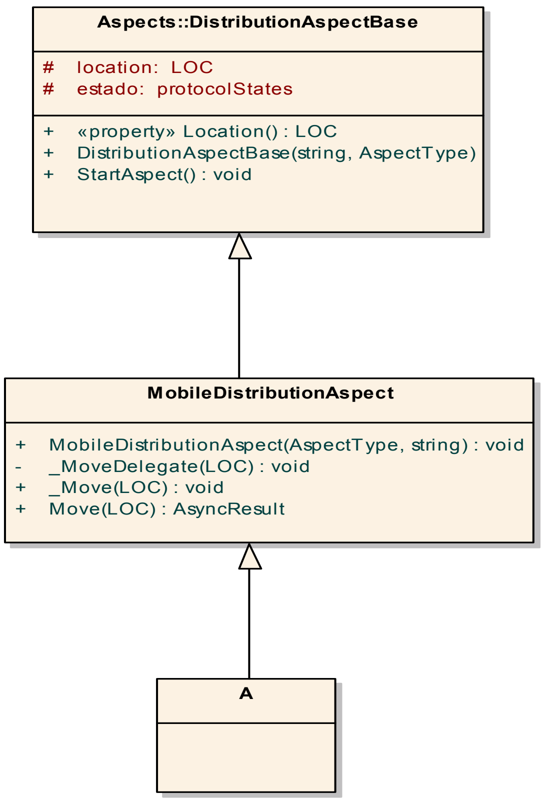

Aspects::DistributionAspectBase
# location: LOC
# estado: protocolStates
+ «property» Location() : LOC
+ DistributionAspectBase(string, AspectType)
+ StartAspect() : void
MobileDistributionAspect
+ MobileDistributionAspect(AspectType, string) : void
- _MoveDelegate(LOC) : void
+ _Move(LOC) : void
+ Move(LOC) : AsyncResult
A

Figure 28 A MobileDistributionAspect is implemented that is specialized from DistributionAspectBase

Participants.

In Figure 26:

- *A* is a distribution aspect which inherits from *MobileDistributionAspec* to become a distribution aspect with the service move implemented.
- *MobileDistributionAspect* inherits from *DistributionAspectBase* to include its own distribution properties including the implementation of the service move that all distribution aspects should specialize to automatically generate the code.
- *DistributionAspectBase* inherits from *AspectBase* to include its own distribution properties that all distribution aspects should specialize to automatically generate the code.

Consequences.

- Facilitates automatic generation of distribution aspects with service move. To make a PRISMA distribution aspect with service move it has to inherit from *MobileDistributionAspect.*
- Good maintainability.
- Reusability. Any distribution aspect to be generated with service move

has to inherit from *MobileDistributionAspect*.

Example.

Figure 29, shows how a distribution aspect called *ExtMbile* has to inherit from *DistributionAspectBase* to be generated.

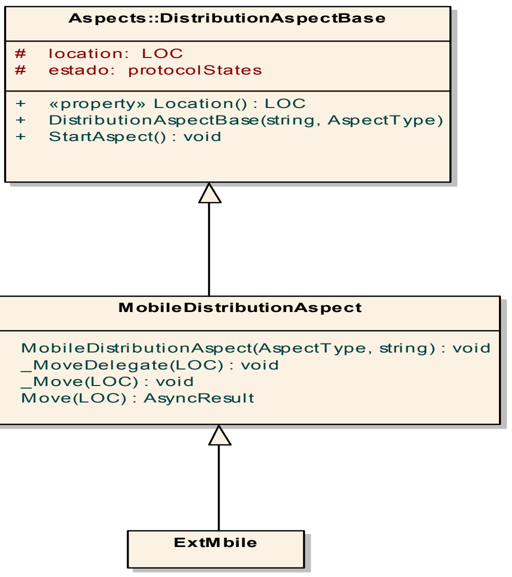

Aspects::DistributionAspectBase
# location: LOC
# estado: protocolStates
+ «property» Location() : LOC
+ DistributionAspectBase(string, AspectType)
+ StartAspect() : void
MobileDistributionAspect
+ MobileDistributionAspect(AspectType, string) : void
- _MoveDelegate(LOC) : void
+ _Move(LOC) : void
+ Move(LOC) : AsyncResult
ExtMbile

Figure 29 A *ExtMbile* distribution aspect has to inherit from the *DistributionAspectBase*

Implementation.

- *public* **MobileDistributionAspect***(string name, AspectType type)*
  : base(name , type)

This is the constructor of MobileDistributionAspect class which has to use the constructor of the MobileDistributionAspect because it has to use the constructor of its base class. The name of the distribution aspect and the type are the arguments of the constructor. The type argument is to indicate that it is a distribution aspect. The code is as following:

```
        public MobileDistributionAspect(): base(name ,
type)     {

            }
```

- *public* ***Move****(LOC)* :AsynchResult

This method is invoked when the component is requested to move. First a delegate is created, then the delegate is queued in the aspect queue. The code is as following:

```
            public AsyncResult Move(LOC newLoc) {
                  _MoveDelegate moveDelegate = new
_MoveDelegate(this._Move);
                  AsyncResult result = new AsyncResult(1);
                  ClassQueueAspect classQueueAspect = new
ClassQueueAspect(moveDelegate,
                        new object[] {newLoc},
System.Threading.Thread.CurrentThread, result);

      queuecallaspect.Enqueue(classQueueAspect);
                  return result;
            }
```

• *public* ***_Move****(LOC)* :void

This method is the internal method of move. When this service is activated the method calls to the component that is going to move. Then it calls asynchronously to the middleware to complete with the mobility process.

```
            private delegate void _MoveDelegate(LOC
newLoc);
            public void _Move(LOC newLoc) {

                  if (estado != protocolStates.MOVEMENT) {
                        throw new
InvalidProtocolStateException(this.aspectName, "Move");
                  }

                  link.IsStopping = true;

                  MiddlewareSystem.MoveDelegate delegado =
new MiddlewareSystem.MoveDelegate(middlewareRef.Move);
                  IAsyncResult iAR =
delegado.BeginInvoke(newLoc, link.componentName,
                        new
AsyncCallback(MiddlewareSystem.MoveCallBack), delegado);

                  middlewareRef.AddMessage("Location of " +
aspectType.ToString() + " aspect has changed to: "
                        + location);

 }
```

Related Patterns.

No related Patterns

3.3.4 Attachments.

Context.

Attachments in PRISMA are responsible of the communication channel between two architectural elements. The PRISMA *attachments* have to be implemented to automatically generate their code. The *attachments* have to be implemented in C# .Net.

Problem.

How is an *attachment* between two architectural elements implemented in C#? How can a PRISMA *attachment* be automatically generated in C# .Net?

Forces.

- Maintenance effort: very low maintenance costs are desirable. The impact of modifying an attachment specification using PRISMA languages and UML notation should be minimum.
- Reusability: The solution proposed should be reusable for all attachments.
- Flexibility: the solution proposed should be adaptable to work with all PRISMA attachments in C#.

Solution.

Two C# classes are necessary to automatically generate a PRISMA attachment between two architectural elements in C#: one to represent the server-side of a partner of an attachment and another to represent the client-side of a partner of an attachment. A PRISMA attachment in C# is implemented into four classes: a server attachment class for each end and a client attachment class for each end (architectural element). Each superclass is associated with a class which is used by the middleware to have a reference of each client-server side of a partner of an attachment.

Structure.

Figure 30, shows a generic class diagram which uses the solution proposed. The *AttachmentClientBase* and *AttachmentServerBase* are responsible of the generation of an attachment. For each partner of an attachment, two classes have to be generated: one must inherit from *AttachmentClientBase* and another must inherit from *AttachmentServerBase*.

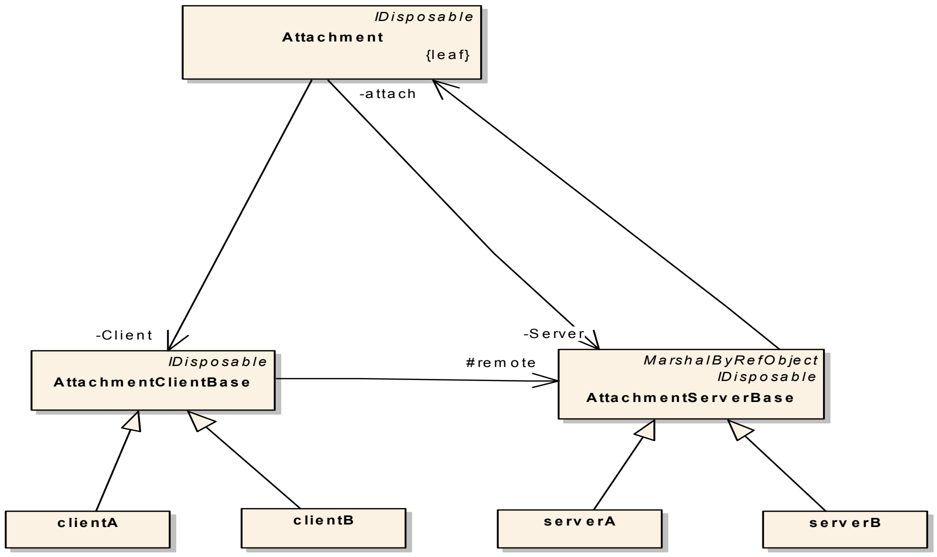


Figure 30 The generation of a PRISMA attachment in C#.

Participants.

- *clientA* is the client part of the attachment of an architectural element A.
- *serverA* is the server part of the attachment of an architectural element A.
- *clientB* is the client part of the attachment of an architectural element B.
- *serverB* is the server part of the attachment of an architectural element B.
- *AttachmentClientBase* is the class that gives the properties necessary to the client part of an attachment once generated. This class is associated with *AttachmentServerBase* to have a reference to which server side it is connected. In addition, it is associated with class *Attachment* to have a reference to which attachment it belongs.
- *AttachmentServerBase* is the class that gives the properties necessary to the server part of an attachment once generated. This class is associated with *AttachmentClientBase* to have a reference to which client side it is connected. In addition, it is associated with class *Attachment* to have a reference to which attachment it belongs. It uses *MarshalByRef* to enable remote communication between the client-side of a partner and the server-side of the other partner of an attachment.
- *Attachment* is a class that has the reference of both sides of an attachment: a client side and a server side of a participant of an attachment.

Consequences.

- Facilitates automatic generation of an attachment. To make a PRISMA attachment four classes have to be generated two inheriting from *AttachmentClientBase* and two inheriting from *AttachmentServerBase*.
- Good maintainability.
- Reusability. Any attachment to be generated has to inherit from *AttachmentServerBase* and *AttachmentClientBase*.

Example.

Figure 31, shows an example of a PRISMA attachment between two components. As attachments in PRISMA are bidirectional, a separation between the client-side and server-side of each part of the attachment has been necessary at implementation.

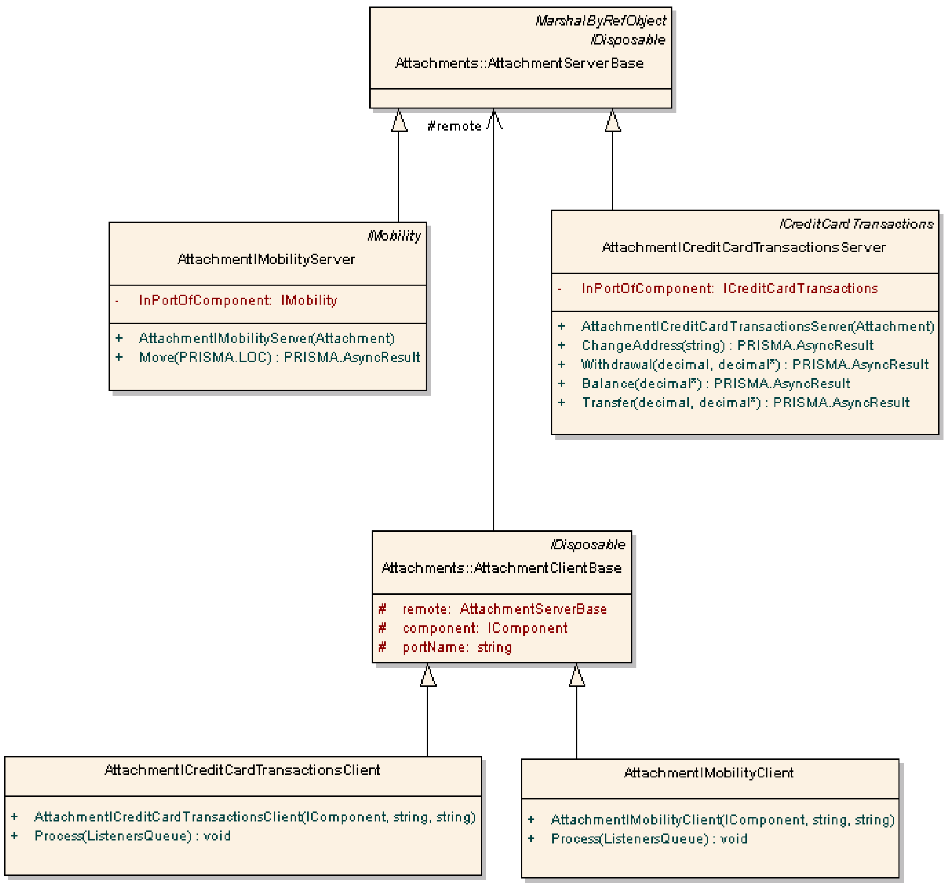

MarshalByRefObject
IDisposable
Attachments::AttachmentServerBase
#remote
IMobility
AttachmentIMobilityServer
- InPortOfComponent: IMobility
+ AttachmentIMobilityServer(Attachment)
+ Move(PRISMA.LOC) : PRISMA.AsyncResult
ICreditCardTransactions
AttachmentICreditCardTransactionsServer
- InPortOfComponent: ICreditCardTransactions
+ AttachmentICreditCardTransactionsServer(Attachment)
+ ChangeAddress(string) : PRISMA.AsyncResult
+ Withdrawal(decimal, decimal*) : PRISMA.AsyncResult
+ Balance(decimal*) : PRISMA.AsyncResult
+ Transfer(decimal, decimal*) : PRISMA.AsyncResult
IDisposable
Attachments::AttachmentClientBase
# remote: AttachmentServerBase
# component: IComponent
# portName: string
AttachmentICreditCardTransactionsClient
+ AttachmentICreditCardTransactionsClient(IComponent, string, string)
+ Process(ListenersQueue) : void
AttachmentIMobilityClient
+ AttachmentIMobilityClient(IComponent, string, string)
+ Process(ListenersQueue) : void

Figure 31 A PRISMA attachment generating four classes: two inheriting from *AttachmentClientBase* and two inheriting from *AttachmentServerBase*.

Figure 32, shows the result of generating the classes in Figure 31. For each partner of the attachment, that is, for each architectural element participating in the attachment part of the attachment is created. Each part of the attachment has its client and server side. The client-side of a participant of an attachment is connected with the server-side of its partner.

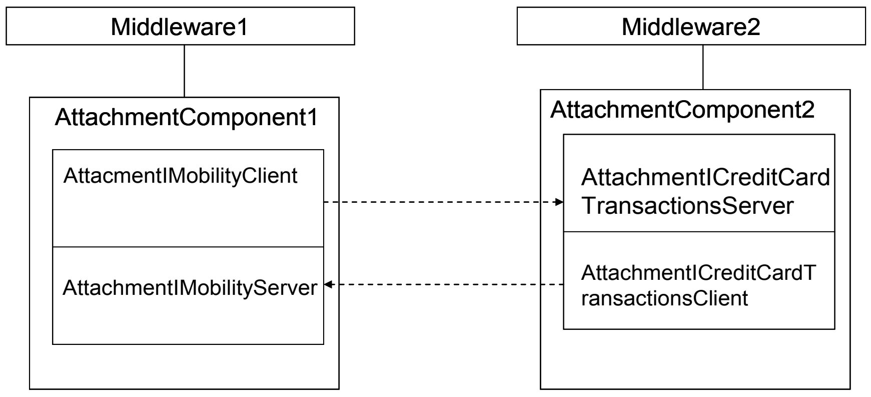

Middleware1
Middleware2
AttachmentComponent1
AttachmentComponent2
AttacmentIMobilityClient
AttachmentIMobilityServer
AttachmentICreditCard TransactionsServer
AttachmentICreditCardT ransactionsClient

Figure 32 A diagram which explains the result of the example of Figure 31.

Implementation.

-*AttachmentClientBase* class is implemented as follows:

- *public* ***AttachmentClientBase****(IComponent component string portName, string attachmentName)*

This is the constructor of the *AttachmentClientBase* class which is invoked by the *Attachment*. It has three parameters: IComponent component, string portName and string attachmentName. IComponent component is the name of the component instance where the attachment will listen. String portName is the name of the port where the attachment will listen. String attachmentName is the name of the global attachment it belongs to. The code is as following:

```
            public AttachmentClientBase(IComponent
component, string portName, string attachmentName) {

                  this.component = component;
                  this.portName = portName;
                  this.attachmentName = attachmentName;
            }
```

- *public* ***AttachmentStart***():void

This method is executed by the *Attachment* class(the aggregate class) to assign the *ClientAttachment* a component port to listen. initiate the thread of the client part. Before executing this method, the method

ConnectToRemoteAttachment must have been called to indicate the reference of the server part it needs to connect with. This service waits until a service is queued to process it. The code is as following:

```
             public void AttachmentStart() {
                  this.queue =
((PRISMA.Components.Ports.OutPort)component.OutPorts[portNa
me]).RegisterListener(attachmentName);

                  hasFinish = false;

                  if (this.remote == null)
                        throw new Exception("The
AttachmentServer reference to Connect to must be instanced
before calling " +
                              "this method, with calling
\"ConnectToRemoteAttachment\" method");

                  isfinish = false;
                  attachmentThread =
System.Threading.Thread.CurrentThread;
                  while(!isfinish) {
                        while( (queue.Count == 0) &&
(!isfinish))

      System.Threading.Thread.Sleep(100);
                        if (!isfinish)

                              Process((ListenersQueue)
queue.Dequeue());
                  }
                  hasFinish = true;
            }
```

- *public* ***ConnectToRemoteAttachment****(* *AttachmentServerBase remote)*:void

This method is invoked by the *Attachment* to indicate to the client attachment the proxy to its server side attachment if they are distributed or a reference to its server side if they are local. The code is as following:

```
            public void
ConnectToRemoteAttachment(AttachmentServerBase remote) {
                  this.remote = remote;
            }
```

- *public* ***AttachmentStop***():void

This method stops the listening to petitions by destroying the thread and unregistering it from the component port it was listening at. The code is as following:

```
            public void AttachmentStop() {
                  isfinish = true;
                  while ( this.hasFinish == false) {
```

```
                                System.Threading.Thread.Sleep(25);
                        }


         ((OutPort)component.OutPorts[portName]).UnRegister(th
is.attachmentName);
                }
```

- *public* ***Dispose****()*:void

This method prepares to destroy the client part. The code is as following:

```
               public void Dispose() {
                }
```

- *public* ***Process****(ListenersQueue queue)*:void

This method processes a petition which is encapsulated in an object of type *ListenersQueue* .The code is as following:

```
               public virtual void Process(ListenersQueue
queue) {}
```

-*AttachmentServerBase* has to be *MarshalByRef.* It is a published class in Remoting. The class is implemented as follows:

- *public* ***AttachmentServerBase****(Attachment attach)*

This is the constructor of the *AttachmentServerBase* class. It has an argument *Attachment attach,* to have a reference to the global attachment. This is necessary because the server side of the attachment has to redirect some necessary petitions to the *Attachment.* These situations are necessary due to the fact that the server side is remotely published and acts as a coordinator between the remote objects and the *Attachment*. The code is as following:

```
               public AttachmentServerBase(Attachment attach) {
                        this.attach = attach;

                }
```

- *protected* ***Component****()*:IComponent

This is a property which enables to access a component to redirect to it the petitions. The code is as following:

```
                protected IComponent Component { get { return
```

```
attach.Component; } }
```

- *public* ***GiveMeName****()*:string

This method returns the name of the component to which the attachment acts as a server proxy. The code is as following:

```
            public string GiveMeName() {
                  return
((PRISMA.Components.IComponent)attach.Component).componentN
ame;
            }
```

- *public* ***GiveMePort****()*:string

This method returns the name of the port to which this part of the client listens on this part of the attachment. The code is as following:

```
           public string GiveMePort() {
                  return attach.PortName;
            }
```

- *public* ***IsPort****()*:bool

This method enables to differeentaiate if it is on a component “true” or a connector “false” side. The code is as following:

```
            public bool IsPort() {
                  return attach.IsPort;
            }
```

- *public* ***MyCoupleName****()*:string

This method returns the name of the other part of the attachment (the client part it is connected with). The code is as following:

```
          public string MyCoupleName() {
                  return attach.MyCoupleName;
            }
```

- *public* ***AttachmentStart****()*:void

This method is called by the *middleware*. The attachment server redirects the method to the *Attachment* so that it passes the proxy of the server side connected to the client attachment to the client attachment. The code is as following:

```
          public void AttachmentStart() {

     attach.AttachmentStart(attach.CreateProxyOfMyCouple()
);
           }
```

- *public* ***AttachmentStop****()*:void

This method is executed by the middleware to stop the attachment The code is as following:

```
           public void AttachmentStop() {
                attach.AttachmentStop();
           }
```

- *public* ***Dispose***():void

This method disposes the part of the attachment. The code is as following:

```
          public void Dispose() {
           }
```

- *public* ***ChangeLocationOfMyCouple****(string couplePath, AttachmentServerBase attachmentServerCouple)*:void

This method notifies the couple of the server attachment of a change in the location of the AttachmentServer. This method makes its couple to change its pointer in cases of mobility.

```
          public void ChangeLocationOfMyCouple(string
couplePath, AttachmentServerBase attachmentServerCouple) {

     attach.ChangeLocationOfMyCouple(couplePath,
attachmentServerCouple);
           }
```

- *public override* ***InitializeLifetimeService****()*:object

This method overrides the remoting service *InitializeLifetimeService()*.

The method returns null indicating that the lifetime of the object is infinite. The code is as following:

```
            public override object
InitializeLifetimeService() {
                  return null;

            }
```

- *public* ***IsLive()***:bool

This method returns "true" if the thread is a live. The code is as following:

```
            public bool IsLive() { return true;}
```

Related Patterns.

No related Patterns

3.3.5 Bindings.

Context.

Bindings in PRISMA are responsible of the communication between a system architectural element and the architectural elements they contain. The PRISMA *bindings* have to be implemented to automatically generate their code. The *bindings* have to be implemented in C# .Net.

Problem.

How is a *binding* between a system and an architectural element it contains implemented in C#? How can a PRISMA *binding* be automatically generated in C# .Net?

Forces.

- Maintenance effort: very low maintenance costs are desirable. The impact of modifying a binding specification using PRISMA languages and UML notation should be minimum.
- Reusability: The solution proposed should be reusable for all bindings.
- Flexibility: the solution proposed should be adaptable to work with all PRISMA attachments in C#.

Solution.

To automatically generate a binding between a PRISMA system and an architectural element contained in a system, only the binding side of the architectural elements contained have to generated. However, the binding side of the system is generated automatically at the generation of a system architectural element. Therefore, two C# classes are necessary to be generated: one to represent the server-side of the binding and another to represent the client side of a binding of an architectural element contained in a system.

Structure.

Figure 33, shows a generic class diagram which uses the solution proposed. The *ComponentBindingClientBase* and *ComponentBindingServerBase* are responsible of the generation of the bindings part of an architectural element contained in a system. For each architectural element that has bindings, two classes have to be generated: one must inherit from *ComponentBindingClientBase* and another must inherit from *ComponentBindingServerBase*. Remember, that the bindings part of the system are automatically generated at the generation of a system architectural element.

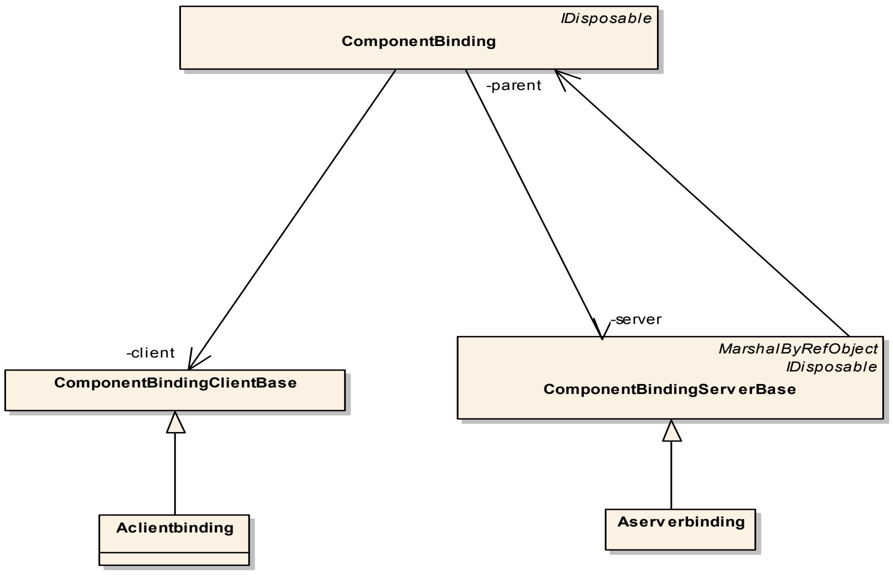


Figure 33 The generation of a PRISMA binding for an architectural element contained in a system in C#.

Participants.

- *AclientBinding* is the client part of the binding of an architectural element A.
- *AserverBinding* is the server part of the architectural element A which participates in the PRISMA attachment.
- *ComponentBindingClientBase* is the class that gives the properties necessary for the client part of a binding of an architectural element contained in a system once generated.
- *ComponentBindingServerBase* is the class that gives the properties necessary to the server part of an architectural elment bindinding once generated. This class is associated with *ComponentBinding* to redirect to it some calls. It uses *MarshalByRef* to enable remote communication such that it is the only published part of a binding.
- *ComponentBinding* is a class that has the reference of both sides of a binding: a client side and a server side of a participant of a binding.

Consequences.

- Facilitates automatic generation of a binding. To make a PRISMA binding of an architectural element contained in a system two classes have to be generated one inheriting from *ComponentBindingClientBase* and the other inheriting from *ComponentBindingServerBase*.
- Good maintainability.

- Reusability. Any binding to be generated has to inherit from *ComponentBindinServerBase* and *ComponentBindingClientBase*.

Example.

Figure 34, shows an example of a binding of an architectural element. As a binding between the system and its architectural element is bidirectional, a separation between the client-side and server-side of the binding side of an architectural element has been necessary at implementation.

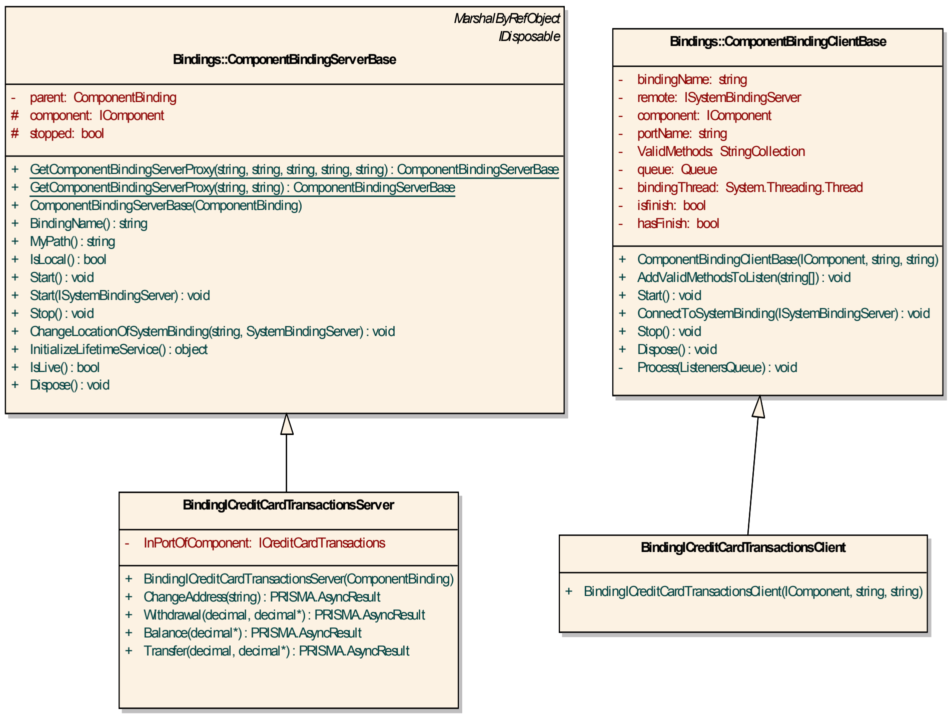

MarshalByRefObject
IDisposable
Bindings::ComponentBindingServerBase
- parent: ComponentBinding
# component: IComponent
# stopped: bool
+ GetComponentBindingServerProxy(string, string, string, string, string) : ComponentBindingServerBase
+ GetComponentBindingServerProxy(string, string) : ComponentBindingServerBase
+ ComponentBindingServerBase(ComponentBinding)
+ BindingName() : string
+ MyPath() : string
+ IsLocal() : bool
+ Start() : void
+ Start(ISystemBindingServer) : void
+ Stop() : void
+ ChangeLocationOfSystemBinding(string, SystemBindingServer) : void
+ InitializeLifetimeService() : object
+ IsLive() : bool
+ Dispose() : void
Bindings::ComponentBindingClientBase
- bindingName: string
- remote: ISystemBindingServer
- component: IComponent
- portName: string
- ValidMethods: StringCollection
- queue: Queue
- bindingThread: System.Threading.Thread
- isfinish: bool
- hasFinish: bool
+ ComponentBindingClientBase(IComponent, string, string)
+ AddValidMethodsToListen(string[]) : void
+ Start() : void
+ ConnectToSystemBinding(ISystemBindingServer) : void
+ Stop() : void
+ Dispose() : void
- Process(ListenersQueue) : void
BindingICreditCardTransactionsServer
- InPortOfComponent: ICreditCardTransactions
+ BindingICreditCardTransactionsServer(ComponentBinding)
+ ChangeAddress(string) : PRISMA.AsyncResult
+ Withdrawal(decimal, decimal*) : PRISMA.AsyncResult
+ Balance(decimal*) : PRISMA.AsyncResult
+ Transfer(decimal, decimal*) : PRISMA.AsyncResult
BindingICreditCardTransactionsClient
+ BindingICreditCardTransactionsClient(IComponent, string, string)

Figure 34 A binding part of an architectural element contained in a system generated by two classes: one inheriting from *ComponentBindingClientBase* and another inheriting from *ComponentBindingServerBase*.

Variants.

None.

Implementation.

-*ComponentBindingClientBase* class is implemented as follows:

- *public* ***ComponentBindingClientBase****(IComponent component, string portName, string bindingName)*

This is the constructor of the *ComponentBindingClientBase* class. The

ComponentBinding instantiates the *ComponentBindingClientBase.* It has three parameters: IComponent component, string portName and string bindingName. IComponent component is the name of the component instance where the binding is connected to. String portName is the name of the port where the binding is connected to. String bindingName is the name of the global binding it belongs to. The code is as following:

```
            public ComponentBindingClientBase(IComponent
component, string portName, string bindingName) {
                  this.component = component;
                  this.portName = portName;
                  this.bindingName = bindingName;
                  this.ValidMethods = new
StringCollection();
            }
```

- *public* **AddValidMethodsToListen**(string[] methods):void

This method has as a parameter the list of methods that the *ComponentBindingClientBase* has to listen to and process from the queue. The code is as the following:

```
            public void AddValidMethodsToListen(params
string[] methods) {
                  ValidMethods.AddRange(methods);
            }
```

- *public* **Start**():void

This method is called by the *ComponentBinding* to initiate the execution of the *ComponentBindingClientBase* thread. The method ConnectToRemoteBinding has to be previously called to indicate the reference of the serverBinding which it has to connect to. The method is registered in an object port to assign a position in the queue where it has to listen. The code is as following:

```
            public void Start() {
                  hasFinish = false;
                  this.queue =
((PRISMA.Components.Ports.OutPort)component.OutPorts[portNam
e]).RegisterListener(bindingName);

                  if (this.remote == null)
                        throw new Exception("The
SystemBinding reference to Connect to must be instanced
before calling " +
                              "this method, with calling
\"ConnectToRemoteBinding\" method");
```

```
                        isfinish = false;
                        bindingThread =
System.Threading.Thread.CurrentThread;
                        while(!isfinish) {
                                while( (queue.Count == 0) &&
(!isfinish))

        System.Threading.Thread.Sleep(100);//
                                if (!isfinish)
                                        Process((ListenersQueue)
queue.Dequeue());
                        }
                        hasFinish = true;
                }
```

- *public* **ConnectToSystemBinding**(ISystemBindingServer remote):void

This method is called to indicate which reference of the binding side of the system it has to connect with to redirect the services. If the binding side of the system is remote it indicates the proxy. The code is as following:

```
                public void
ConnectToSystemBinding(ISystemBindingServer remote) {
                        this.remote = remote;
                }
```

- *public* **Stop**():void

This method stops the ComponentBindingClientBase instance from listening to petitions by destroying the thread and unregistering it from the component it was listening at. The code is as following:

```
                public void Stop() {
                        isfinish = true;
                        while ( this.hasFinish == false) {
                                System.Threading.Thread.Sleep(25);
                }

        ((OutPort)component.OutPorts[portName]).UnRegister(thi
s.bindingName);
                }
```

- *public* ***Dispose***()*:void

This method prepares to destroy the client part. The code is as following:

```
              public void Dispose() {
                }
```

- *public* ***Process****(ListenersQueue queue)*:void

This method processes a petition which is encapsulated in an object of type

`ListenersQueue`. The petition is checked if it corresponds to a method implemented by this interface. If it is a method implemented by the interface then the petition is redirected to the *SystemBinding*. The code is as following:

```
            private void Process(ListenersQueue queue) {
                   if
(ValidMethods.Contains(queue.NombreServicio)) {
                         AsyncResult result =
remote.AddRequest(this.bindingName, queue.NombreServicio,
queue.Args);

      queue.Result.SetDerivedParameters(result);
                   }
             }
```

-*ComponentBindingServerBase* has to be *MarshalByRef.* The class is implemented as follows:

- *public* **ComponentBindingServerBase**(ComponentBinding compBind):

This is the constructor of the *ComponentBindingServerBase* class. It has an argument *ComponentBinding compBind,* to have a reference to the global ComponentBinding to allow it to redirect some petitions. The code is as following:

```
            public
ComponentBindingServerBase(ComponentBinding compBind) {
                   this.parent = compBind;

             }
```

- *public static* **GetComponentBindingServerProxy**(string systemName, string systemPort, string componentName, string portName, string componentPath):ComponentBindingServerBase

This method has as parameters the properties that define a binding between a system and an architectural element. These are necessary because using this information the *ComponentBindingServerBase* creates a proxy. The code is as following:

```
             public static ComponentBindingServerBase
GetComponentBindingServerProxy(
                              string systemName, string
systemPort, string componentName, string portName,
                              string componentPath)
            {
                              string ComponentBindingName =
"ComponentBinding[" + systemName + "]" +
                        systemPort + "[" + componentName +
      "]" + portName;
                  if (!componentPath.StartsWith("tcp://") )
                        componentPath = "tcp://" +
componentPath;
                  string URL = componentPath + ":" +
MiddlewareSystem.serverPort + "/" + ComponentBindingName +
".soap";
                  return (ComponentBindingServerBase)
Activator.GetObject(typeof(ComponentBindingServerBase),
URL);
            }
```

- *public **BindingName**():string*

This method returns the name of the *ComponentBinding* it belongs to. The code is as following:

```
            public string BindingName() {
                  return parent.Name;
            }
```

- *public* **MyPath**():string

This method returns the path it is located at. The code is as following:

```
           public string MyPath() {
                  return parent.MyPath;
            }
```

- *public* **IsLocal**():bool

This method returns if it is local or distributed to the system it is connected to. The code is as following:

```
            public bool IsLocal() {
                  return parent.IsLocal;
      }
```

- *public* **Start**():void

This method is used to be redirected to the *ComponentBinding* to start the thread of the *ComponentBindingClientBase*. This method is used when the communication is remote to pass the proxy of the *SystemBinding*. The code is as following:

```
          public void Start() {
```

```
      parent.Start(parent.CreateProxyOfSystemBinding());
            }
```

- *public* **Start**(ISystemBindingServer systemBindingServer):void

This method is called when the communication is local with the binding side of the system. The method initiates the *ComponentBinding* passing the reference of the *SystemBinding*. The code is as following:

```
          public void Start(ISystemBindingServer
systemBindingServer) {
                  this.stopped = false;
                  if (parent.IsLocal)
                        parent.Start(systemBindingServer);
                  else
                        this.Start();
            }
```

- *public* **Stop**():void

This method calls the Stop method of the *ComponentBinding*. The code is as following:

```
            public void Stop() {
                  parent.Stop();
                  this.stopped = true;

      }
```

- *public* **ChangeLocationOfSystemBinding**(string systemPath, SystemBindingServer bindingServer):void

This method is called when the system changes its location. The *ComponentBindingServerBase* notifies *ComponentBinding* the new path of the binding of the system. The code is as following:

```
           public void ChangeLocationOfSystemBinding(string
systemPath, SystemBindingServer bindingServer) {

      parent.ChangeLocationOfSystemBinding(systemPath,
bindingServer);
            }
```

- *public* **InitializeLifetimeService**():object

This method overrides the remoting service *InitializeLifetimeService()*. The method returns null indicating that the lifetime of the object is infinite. The code is the following:

```
          public override object InitializeLifetimeService()
```

```
{
                    return null;
          }
```

- *public* ***IsLive()***:bool

This method returns “true” if the thread is a live. The code is as following:

```
          public bool IsLive() { return true;}
```

- *public* ***Dispose***():void

This method disposes the part of the attachment. The code is as following:

```
      public void Dispose() {
       }
```

Related Patterns.

No related Patterns

CHAPTER 4. CONCLUSIONS AND FUTURE WORKS

As a conclusion, this chapter sums up the main contributions of the work, scientific publications and gives some suggestions for further work.

4.1 Summary of the Contributions

The research of this work contributes as a first step in reaching the objective of automatically generating distributed applications from a conceptual model. The MDA proposal has been applied at all levels. A Platform Specific Model of the PRISMA distribution model has been presented. Thus the attachments, binding and mobility specific models have been described.

Next, the implementation patterns have been identified to automatically generate distributed applications from the PRISMA distribution conceptual model. The implementation patterns describe which base classes of the PRISMA distribution model implemented in C# have to be extended to generate final distributed applications.

As a result of this work, a first step in building a model compiler for the PRISMA architectutral model has been satisfied. Thus PRISMA becomes a framework to describe distributed systems at an analysis and design level (conceptual level).

This work has been financed by a Microsfoft Research project for implementing a teleoperation system using PRISMA. Thus, this work

4.2 Related Publications

This work is based on a set of research publications. The following international and national publications were obtained:

- **Nour Ali,** Jose M. Cercos, Isidro Ramos, Patricio Letelier, Jose A. Carsi. “Distribution in PRISMA” Actas de II jornadas de trabajo de DYNAMICA, Malaga, Noviembre 2004. (To appear)
- **Nour Ali,** Jennifer Perez, Cristobal Costa, Jose A. Carsi, Isidro Ramos. “Implementation of the PRISMA Model in the .Net Platform”, Actas de II jornadas de trabajo de DYNAMICA, Malaga, Noviembre 2004. (To appear)

4.3 Further Work

In the near future the research is going to be concentrated on how to implement the transformation patterns identified in this research. Some tools are going to be necessary to implement them. After implementing the patterns a model compiler is going to build to incorporate them to automatically generate the code from our specification model.

In addition, another important task is implementing the teleoperation case study using the C# PRISMA model presented in this work. This case study is going to be used to test the performance of our proposal.

Another important work is to implement the Distribution patterns discussed in [3]. These patterns to be implemented in C# are going to increase the PRISMA middleware functionality in distribution matters.

In addition, a case tool is going to be developed to incorporate textual and graphical notation of PRISMA and patterns to enable develop the distributed applications in different platforms and programming languages.

BIBLIOGRAPHY

[1] **Ali, N.**, Carsi, J.A., Ramos, I. Analysis of a Distribution Dimension for PRISMA. Actas Jornadas de Ingeniería del Software y Bases de Datos, JISBD, Malaga. (To Appear Accepted as short paper)

[2] Ali, N.H., Perez J., Ramos I. "High Level Specification of Distributed and Mobile Information Systems", Proceedings of Second International Symposium on Innovation in Information & Communication Technology ISIICT 2004, Amman, Jordan, 21-22 April, 2004.

[3] Ali, N.H., Silva J., Jaen, J.,Ramos I., Carsi, J.A., and Perez J. Mobility and Replicability Patterns in Aspect-Oriented Component-Based Software Architectures. Proceedings of 15th IASTED, Parallel and Distributed Systems, Acta Press (Marina del Rey, C.A., USA, November 2003), p 820-826.

[4] Aspect-Oriented Software Development, http://aosd.net

[5] Archer,T. and WhiteChapel, A. Inside C# second Edition. Copyright © 2002 by Microsoft Corporation

[6] Balzer, R. Software Tech. in the 1990's: Using a new Paradigm", IEEE, 1983.

[7] Bernstein, P.A. Middleware: a model for distributed system services. Communications of the ACM, Volume 39, Issue 2, February 1996 ISSN:0001-0782, p 86-98.

[8] Budinsky, F.J, Finnie, M.A, Vlissides, J.M and Yu, P.S. Automatic code generation from design patterns. IBM Systems Journal. Volume 35, Object technology, Number 2, 1996. http://www.research.ibm.com/journal/sj/352/budinsky.html

[9] CORBA Official Web Site of the OMG Group: http://www.corba.org/

[10] Crupi, J. and Baerveldt, F. Implementing Sun Microsystems' Core J2EE Patterns. Compuware White Paper.

[11] Dhawan, P. Performance Comparison: .NET Remoting vs. ASP.NET Web Services. Building Distributed Applications with Microsoft .NET. September 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdadotnetarch14.asp

[12] Dhawan, P. and Ewald, T. ASP.NET Web Services or .NET Remoting: How to Choose Building Distributed Applications with Microsoft .NET. September 2002. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdadotnetarch16.asp

[13] Fuggetta, A., Picco, G.P., and Vigna, G. Understanding Code Mobility. In *IEEE Transactions on Software Engineering*, 24(5): 342-361, 1998.

[14] Grau A., "*Computer-Aided validation of formal conceptual models*", PhD. Thesis Institute for Software, Information Systems Group, Technical University of Braunschweig, March 2001.

[15] MDA web page: www.omg.com/mda

[16] Microsoft .Net Remoting : A Technical Overview, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp

[17] Oblog – Object Logic. "*Oblog Software*", Oblog Software S.A. Lisboa, Portugal. URL: http://www.info.fundp.ac.be/~phe/2rare/spd.html

[18] Pastor O. Et al, *OO-METHOD: A Software Production Environment Combining Conventional and Formal Methods*, Procc. of 9th International Conference, CaiSE97, Barcelona, 1997.

[19] Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures: *3rd IEEE International Conference on Quality Software (QSIC 2003)*, Dallas, Texas, USA, November 2003, p 59-66.

[20] Polo, M, Piattini, M. and Ruiz, F. Reflective Persistence (Reflective CRUD: Reflective Create, Read, Update & Delete). Proceedings of Conference EuroPlop 2001.

[21] Rammer, I. Advanced .Net Remoting. A press; 1 edition (April 5, 2002), ISBN: 1590590252.

[22] Rational Software, Rational Rose, http://www.rational.com/products/rose/

[23] Sernadas A., Costa J.F., Sernadas C., "Object Specifications Through Diagrams: OBLOG Approach" INESC Lisbon 1994.

[24] Siegel, J. and the OMG Staff Strategy Group. Developing in OMG's Model-Driven Architecture.Object Management Group White Paper, November, 2001.

[25] StrawMyer, M. .Net Remoting. http://www.developer.com/net/cplus/article.php/10919_1479761_1

[26] Szyperski, C., *Component software: beyond object-oriented programming*, (New York, USA: ACM Press and Addison Wesley, 2002).

[27] Troger, P. and Polze, A. Object and Process Migration in .NET .The Eighth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), January 15 - 17, 2003,Guadalajara, Mexico.

[28] Thangarathnim,T. .NET Remoting Versus Web Services. http://www.developer.com/net/net/article.php/11087_2201701_1

[29] Unified Modelling Language UML 2.0: http://www.omg.org/technology/uml

[30] Web Services Tutorials. See http://www.systinet.com/resources/tutorials

Appendix A : PRISMA MAPPINGS TO C#

This appendix shows the whole mappings of the PRISMA model to C#. The mappings of this appendix have been done in collaboration with **PRISMA: Model Compiler of Aspect-Oriented Component-Based Software Architectures** members (Jennifer Perez, Cristobal Costa, Jose Manuel Cercos and Rafael Cabedo).

PRISMA MODEL	**C#**
Components	Collection Class of Components
Interface	Interface
Connectors	Collection Class Component
Attachments	Collection Class Attachments
Binding	2 classes one for the system and another for the architectural element container in the system. The bindingSystem class acts as a server of the binding of the architectural element contained. The other class for implementing the binding side of the architectural element is composed of a client and a Server side.
System	System Class

COMPONENT

MODEL	**C #**
Name	Variable
Ports	Collection of ports of a component. It is divided into a client and a Server.
Interface	A set of services that the middleware invokes to manage the components.
Component	A base class with referentes to aspects, a proper thread, ports, weavings, a queue for the services. Queue= 2 classes, one for the nodes and another for the priorities.
Aspects	Variables with the aspects
Weavings	Weaving Collection Clase = List of Weavings associated to a component Weaving Class Node = List of weavings associated to the service which triggers the weaving Weaving Method Class= Class which contains the service and the

	transformation functions necessary for the weaving parameters. Weaving type Class= Types of weaving Transformation Function= Transformation Functions which pass parameters.

CONNECTOR

MODEL	C #
Name	Variable
Roles	Collection of Ports (= component)
Interface	Subclase of the component interface
Connector	A composite component Class which is identified as a connector for implementing a connector interface (subclass of the component interface)

SYSTEM

MODEL	C #
Name	Variable
Roles	Collection of ports
Interface	Subclass of the component interface ¡
System	A subclass of component which contains the list of components, connectors and attachments and bindings.

ASPECT

MODEL	C #
Aspect Type	An internal variable of the aspect which is assigned a value o fan enumerated type. This type contains the list of aspect types defined in the metamodel.
Name of Aspect	A variable
Aspect	Class with the list of subprocesses, thread, (attributes and services)
Attributes	Variables
Services	2 methods internal and external: external= Queues the service petition and returns the control to the component. Internal= Executes the service
Valuations	Implementation of the internal method to which the valuation is associated. Valuation Condition= if condition Valuation Postcondition = Assigns the values of the internal variables of the if
Preconditions	If of the associated method condition.

	This condition should be executed before of an If valuation
Subprocesses	Class which contains the name of the subprocessed and the set of methods with their priority
Protocol	The set of the protocol states is stored in an enumerated structure If is used to check that the states are correct, for allowing the execution of the methods. The if should be checked before the preconditions associated to a method. The sequence of a state services are implemented with the invocation of the following method. The state is updates at the end of a method.

PORT

MODEL	C #
Port	C# base class without subprocessed with two child classes for seperately treating the client and Server behaviour. Server = Passes the petitions to the component queue Client = Saves the invocation in a queue in which an attachment listens.
Name	Variable with name.
Interface	Variable with interface

ATTACHMENTS

MODEL	C #
Name	Attachment Name
Attachment	2 attachment classes each associated to a side ofthe communication channel through the ports and components. Each attachment class is formed of 2 classes, a client class and another Server to process the the petitions in different channels.

BINDINGS

MODEL	**C #**
Name	Name
Binding	2 classes one for the system the other for the component or connector which forms the system. The bindingsystem class acts as a remote Server for those components which reside on another machine. The class which implements the component and connector is composed.

MIDDLEWARE

PRISMA MODEL	**C #**
PRISMA Metamodel	A class with the evolution services, manages the system and mobility.
LOC	A class which manages the locations of the different architectural elements of the system.
Asynchronous	A clase for managing the asynchronous results of the services.
Exceptions	Exceptions
Services priorities	A queue and list class with priorities.
System Execution	PRISMA Server which starts the execution of the middleware. The Framework initializes the application.

Appendix B: Implementation of a Simple Bank Account using the PRISMA Distribution Model in C#

B.1 Distribution Aspect of an Account

```
using System;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Middleware;

namespace CuentaBancaria {

      /// <summary>Descripción del Aspecto de Distribución
EXTMBILE</summary>
      [Serializable]
      public class ExtMbile : DistributionAspectBase, IMobility {

            #region Definicion de Variables internas

            // Definición de los estados posibles del objeto
            protected enum protocolStates
            {
                  EXTMBILE,
                  MOVEMENT
            }

            // Almacena el estado actual del aspecto
            protected protocolStates estado;

            #endregion

            // Constructor del Aspecto: Inicialización
            public ExtMbile() : base("ExtMbile",
AspectType.Distribution)
            {
                  // Inicializacion de las vbles internas
                  estado = protocolStates.EXTMBILE;

                  // Inicialización de las vbles del aspecto
                  /*if (location.IsNull() ) {// Cuando entre en el
StartAspect tendrá un valor
                        throw new Exception("Location of component
can't be a NULL value");
                  }*/

                  // Creación de la estructura de subprocesos para este
acpecto
                  SubProcessClass subProcessIMobility = new
SubProcessClass("SERVIDOR");
                  subProcessIMobility.AddMethod("Move", 10);
```

```
                    this.subProcessesList.Add(subProcessIMobility);

                    // Alcanzamos el estado siguiente
                    estado = protocolStates.MOVEMENT;
              }

              #region Miembros Externos de IMobility

              public AsyncResult Move(LOC newLoc) {
                    // Encola la petición de mover para que sea servida
cuando acabe las peticiones anteriores
                    _MoveDelegate moveDelegate = new
_MoveDelegate(this._Move);
                    AsyncResult result = new AsyncResult(1);
                    ClassQueueAspect classQueueAspect = new
ClassQueueAspect(moveDelegate,
                          new object[] {newLoc},
System.Threading.Thread.CurrentThread, result);
                    queuecallaspect.Enqueue(classQueueAspect);
                    return result;
              }

              #endregion

              #region Miembros Internos de IMobility

              private delegate void _MoveDelegate(LOC newLoc);
              public void _Move(LOC newLoc) {
                    // Comprobacion de que el protocolo actual es el
correcto
                    if (estado != protocolStates.MOVEMENT) {
                          throw new
InvalidProtocolStateException(this.aspectName, "Move");
                    }

                    // Comprobacion de las precondiciones asociadas
                    //          NO APLICABLE

                    // Comprobación del estado anterior a la valuación
                    //          NO APLICABLE

                    // Indica lo antes posible al componente que va a ser
movido
                    link.IsStopping = true;

                    // Llamada asíncrona al Middleware para que ejecute
el move y nosotros acabar
                    //   con lo que nos queda de este hilo
                    MiddlewareSystem.MoveDelegate delegado = new
MiddlewareSystem.MoveDelegate(middlewareRef.Move);
                    IAsyncResult iAR = delegado.BeginInvoke(newLoc,
link.componentName,
                          new
AsyncCallback(MiddlewareSystem.MoveCallBack), delegado);

                    middlewareRef.AddMessage("Location of " +
aspectType.ToString() + " aspect has changed to: "
                          + location);
```

```
                  // Comprobación de la activación de Triggers
                  //          NO APLICABLE

                  // Nuevo estado alcanzado
                  // estado = protocolStates.MOVEMENT;
            }

            #endregion


            /// <summary>Convierte una dirección postal en una
localización URL</summary>
            public LOC ChangeAddressToLOC(string newAdd) {
                  LOC newLOC;

                  if (newAdd != "") {
                        // Equivalencia de calles a IPs
                        if (newAdd.StartsWith("C/Mayor")) {newLOC = new
LOC("tcp://sigil.dsic.upv.es"); }
                        else {
                              if (newAdd.StartsWith("C/Menor")) {newLOC
= new LOC("tcp://garbi.dsic.upv.es"); }
                              else {
                                    if (newAdd.StartsWith("C/Medio"))
{newLOC = new LOC("tcp://grid001.dsic.upv.es"); }
                                    else {
                                          if
(newAdd.StartsWith("C/Lado")) {newLOC = new
LOC("tcp://poypoy.dsic.upv.es"); }
                                          else {
                                                // Sino, se queda en el
equipo local
                                                newLOC = new
LOC("tcp://localhost");
                                          }

                                    }
                              }
                        }
                  }
                  else { newLOC = new LOC("tcp://localhost"); }

                  return newLOC;
            }
            public delegate LOC ChangeAddressToLOCDelegate(string
newAdd);
      }}
```

B.2 Attachments of the Bank System

```
using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;
```

```
namespace CuentaBancaria {

      /// <summary>
      /// Clase generada automáticamente al generarse el código de la
interfaz "ICreditCardTransactions"
      /// </summary>
      [Serializable]
      public class AttachmentICreditCardTransactionsClient :
AttachmentClientBase {

            /// <summary>
            /// Constructor - Llama al padre
            /// </summary>
            public AttachmentICreditCardTransactionsClient(IComponent
component, string portName, string attachmentName)
                  : base(component, portName, attachmentName) {}


            /// <summary>
            /// Procesa una petición de servicio, encapsulada en un
objeto de tipo ClassQueueAttachment
            /// </summary>
            public override void Process(ListenersQueue queue) {
                  try {
                        // Miramos si la petición de este servicio es
para que el actual attachment o para otro con la misma interfaz
                        // o si es un broadcast
                        //if ((queue.NameRequest == remoteName) ||
(queue.NameRequest == null))
                        switch(queue.NombreServicio) {
                              case "ChangeAddress": {
                                    string arg0 =(string)queue.Args[0];
                                    AsyncResult result =
((ICreditCardTransactions) remote).ChangeAddress(arg0);

      queue.Result.SetDerivedParameters(result);
                                    break;
                              }
                              case "Withdrawal": {
                                    decimal arg0
=(decimal)queue.Args[0];
                                    decimal arg1
=(decimal)queue.Args[1];
                                    Console.WriteLine("[SALIDA] Envio
una peticion de WithDrawal - " + DateTime.Now.Second +
                                          ":" +
DateTime.Now.Millisecond);
                                    AsyncResult result =
((ICreditCardTransactions) remote).Withdrawal(arg0, ref arg1);

      queue.Result.SetDerivedParameters(result);
                                    break;
                              }
                              case "Balance": {
                                    decimal arg0
=(decimal)queue.Args[0];
                                    AsyncResult result =
((ICreditCardTransactions) remote).Balance(ref arg0);
```

```
	queue.Result.SetDerivedParameters(result);
					break;
				}
				case "Transfer": {
					decimal arg0
=(decimal)queue.Args[0];
					decimal arg1
=(decimal)queue.Args[1];
					AsyncResult result =
((ICreditCardTransactions) remote).Transfer(arg0, ref arg1);

	queue.Result.SetDerivedParameters(result);
					break;
				}
				default:
					break;
			}
		}catch( Exception e) {
			throw e;
		}
	}
}
}


using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;

namespace CuentaBancaria {

	/// <summary>
	/// Clase generada automáticamente al generarse el código de la
interfaz "ICreditCardTransactions"
	/// </summary>
	[Serializable]
	public class AttachmentICreditCardTransactionsServer :
AttachmentServerBase, ICreditCardTransactions {

		private ICreditCardTransactions InPortOfComponent;

		/// <summary>
		/// Constructor - Llama al padre
		/// </summary>
		public AttachmentICreditCardTransactionsServer(Attachment
attach) : base (attach) {
			InPortOfComponent = (ICreditCardTransactions)
Component.InPorts[attach.PortName];

			if (InPortOfComponent == null) {
				throw new Exception("ICreditCardTransactions's
InPort does not exists in " +
					Component.componentName + " component.");
			}
		}
```

```
            /* REDIRECCIÓN DE LOS SERVICIOS HACIA EL COMPONENTE DESTINO
*/

            public PRISMA.AsyncResult ChangeAddress(string newAdd) {
                  return InPortOfComponent.ChangeAddress(newAdd);
            }

            public PRISMA.AsyncResult Withdrawal(decimal quantity, ref
decimal money) {
                  Console.WriteLine("[LLEGADA] Recibo una peticion de
WithDrawal - " + DateTime.Now.Second +
                        ":" + DateTime.Now.Millisecond);
                  return InPortOfComponent.Withdrawal(quantity, ref
money);
            }

            public PRISMA.AsyncResult Balance(ref decimal money) {
                  try {
                        return InPortOfComponent.Balance(ref money);
                  } catch ( Exception e) {
                        Console.WriteLine("sdfsd");
                        //throw e;
                        AsyncResult result = new AsyncResult(1);
                        result.ThrowExceptionToClient(e);
                        return result;
                  }

            }

            public PRISMA.AsyncResult Transfer(decimal quantity, ref
decimal money) {
                  return InPortOfComponent.Transfer(quantity, ref
money);
            }
      }
}

using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;

namespace CuentaBancaria {

      /// <summary>
      /// Clase generada automáticamente al generarse el código de la
interfaz "IMobility"
      /// </summary>
      [Serializable]
      public class AttachmentIMobilityClient : AttachmentClientBase {

            /// <summary>
            /// Constructor - Llama al padre
            /// </summary>
            public AttachmentIMobilityClient(IComponent component,
string portName, string attachmentName)
                  : base(component, portName, attachmentName) {}
            /// <summary>
            /// Procesa una petición de servicio, encapsulada en un
objeto de tipo ClassQueueAttachment
            /// </summary>
            public override void Process(ListenersQueue queue) {

                  // Miramos si la petición de este servicio es para
que el actual attachment o para otro con la misma interfaz
                  // o si es un broadcast
                  //if ((queue.NameRequest == remoteName) ||
(queue.NameRequest == null))
                  try{
                        switch(queue.NombreServicio) {
                              case "Move": {
                                    LOC arg0 =(LOC)queue.Args[0];
                                    AsyncResult result = ((IMobility)
remote).Move(arg0);

      queue.Result.SetDerivedParameters(result);
                                    break;
                              }
                              default:
                                    break;
                        }
                  }catch ( Exception e) {
                        throw e;
                  }
            }
      }
}



using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Attachments;

namespace CuentaBancaria {

      /// <summary>
      /// Clase generada automáticamente al generarse el código de la
interfaz "IMobility"
      /// </summary>
      [Serializable]
      public class AttachmentIMobilityServer : AttachmentServerBase,
IMobility {

            private IMobility InPortOfComponent;

            /// <summary>
            /// Constructor - Llama al padre
            /// </summary>
            public AttachmentIMobilityServer(Attachment attach) : base
(attach) {
                  InPortOfComponent = (IMobility)
Component.InPorts[attach.PortName];
```

```
                  if (InPortOfComponent == null) {
                        throw new Exception("IMobility's InPort does
not exists in " +
                              Component.componentName + " component.");
                  }
            }


            /* REDIRECCIÓN DE LOS SERVICIOS HACIA EL COMPONENTE DESTINO
*/

            public PRISMA.AsyncResult Move(PRISMA.LOC newLoc) {
                  // AttachmentAccount.Move no está publicado, por lo
que no se puede acceder
                  return InPortOfComponent.Move(newLoc);
            }
      }
```

B.3 Bindings of the Bank System Example

```
sing System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Bindings;

namespace CuentaBancaria {

      /// <summary>
      /// Clase generada automáticamente al generarse el código de la
interfaz "ICreditCardTransactions"
      /// </summary>
      [Serializable]
      public class BindingICreditCardTransactionsClient :
ComponentBindingClientBase {

            /// <summary>
            /// Constructor - Llama al padre
            /// </summary>
            public BindingICreditCardTransactionsClient(IComponent
component, string portName, string bindingName)
                  : base(component, portName, bindingName) {

                  // Añadimos los métodos que reconocerá este CLIENTE
                  AddValidMethodsToListen("ChangeAddress",
"Withdrawal", "Balance", "Transfer");
            }
      }
}



using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Bindings;

namespace CuentaBancaria {

      /// <summary>
      /// Clase generada automáticamente al generarse el código de la
interfaz "ICreditCardTransactions"
      /// </summary>
      [Serializable]
      public class BindingICreditCardTransactionsServer :
ComponentBindingServerBase, ICreditCardTransactions {

            private ICreditCardTransactions InPortOfComponent;

            /// <summary>
            /// Constructor - Llama al padre
            /// </summary>
            public
BindingICreditCardTransactionsServer(ComponentBinding compBind) : base
(compBind) {
                  InPortOfComponent = (ICreditCardTransactions)
component.InPorts[compBind.PortName];

                  if (InPortOfComponent == null) {
                        throw new Exception("ICreditCardTransactions's
InPort does not exists in " +
                              component.componentName + " component.");
                  }
            }


            /* REDIRECCIÓN DE LOS SERVICIOS HACIA EL COMPONENTE DESTINO
*/

            public PRISMA.AsyncResult ChangeAddress(string newAdd) {
                  if (stopped)
                        throw new
PRISMA.IsMovingComponentException(this.BindingName());
                  return InPortOfComponent.ChangeAddress(newAdd);
            }

            public PRISMA.AsyncResult Withdrawal(decimal quantity, ref
decimal money) {
                  if (stopped)
                        throw new
PRISMA.IsMovingComponentException(this.BindingName());
                  return InPortOfComponent.Withdrawal(quantity, ref
money);
            }

            public PRISMA.AsyncResult Balance(ref decimal money) {
                  if (stopped)
                        throw new
PRISMA.IsMovingComponentException(this.BindingName());
                  return InPortOfComponent.Balance(ref money);
            }
```

```
		public PRISMA.AsyncResult Transfer(decimal quantity, ref
decimal money) {
			if (stopped)
				throw new
PRISMA.IsMovingComponentException(this.BindingName());
			return InPortOfComponent.Transfer(quantity, ref
money);
		}

	}
}


using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Bindings;

namespace CuentaBancaria {

	/// <summary>
	/// Clase generada automáticamente al generarse el código de la
interfaz "IMobility"
	/// </summary>
	[Serializable]
	public class BindingIMobilityClient : ComponentBindingClientBase
{

		/// <summary>
		/// Constructor - Llama al padre
		/// </summary>
		public BindingIMobilityClient(IComponent component, string
portName, string bindingName)
			: base(component, portName, bindingName) {

			// Añadimos los métodos que reconocerá este CLIENTE
			AddValidMethodsToListen("Move");
		}
	}
}



using System;

using PRISMA;
using PRISMA.Components;
using PRISMA.Bindings;

namespace CuentaBancaria {

	/// <summary>
	/// Clase generada automáticamente al generarse el código de la
interfaz "IMobility"
	/// </summary>
	[Serializable]
	public class BindingIMobilityServer :
ComponentBindingServerBase, IMobility {
```

```
            private IMobility InPortOfComponent;

            /// <summary>
            /// Constructor - Llama al padre
            /// </summary>
            public BindingIMobilityServer(ComponentBinding compBind) :
base (compBind) {
                  InPortOfComponent = (IMobility)
component.InPorts[compBind.PortName];

                  if (InPortOfComponent == null) {
                        throw new Exception("IMobility's InPort does
not exists in " +
                              component.componentName + " component.");
                  }
            }


            /* REDIRECCIÓN DE LOS SERVICIOS HACIA EL COMPONENTE DESTINO
*/

            public PRISMA.AsyncResult Move(PRISMA.LOC newLoc) {
                  if (stopped) {
                        throw new
PRISMA.IsMovingComponentException(this.BindingName());
                  }
                  return InPortOfComponent.Move(newLoc);
            }
      }
                              }
```