Modeling Interactions between Web Applications and Third Party Systems

Nathalie Moreno and Antonio Vallecillo
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga, Spain
{vergara,av}@lcc.uma.es

Abstract

Web-based applications are no longer isolated systems. Now they need to interoperate with external service providers and legacy systems, which are available in a wide range of different platforms, and may follow disparate communication mechanisms. Modeling the interactions between these systems is not simple, and need to be properly addressed within any model-driven development scenario. Many of the existing Web Engineering proposals do not take this fact into account, or address it in a very simplistic way. In this work we use an MDA approach for encapsulating the different interaction abstractions and mechanisms into a separate platform-independent level, and show the transformations required to produce platform-specific models depending on the particular details and interaction mechanisms of each technology platform and middleware.

1 Introduction

As the demand and the number of available distributed Web applications grows, so does the need to easily design, deploy, maintain, *integrate* and *interconnect* such Web applications in heterogeneous environments. MDA [18, 5] seems to be one of the most promising approaches for addressing these issues: it provides the right kinds of abstractions and mechanisms for improving the way applications are integrated and interconnected nowadays.

A proper integration approach requires an structured and efficient way to assist software architects and developers achieve such integration not only at implementation level, but also during all phases of the development process. In this regard, any integration with legacy data and external services at the PIM level requires modeling them, too (their structural features, behavioral descriptions, etc.)—allowing the manipulation of the external entities of such systems as native elements of our models. Special care should be taken in this case with the bridges that connect the system with its external partners applications, for which transformations are also needed, as mentioned in [11].

Although a priori there are no major problems with this approach, we may face different kinds of incompatibility issues when trying to integrate external pieces into the system (e.g., external services or legacy applications). For instance, the interface of the services required by our application (as specified in one of the PIMs) may not match the interface of the actual service, as provided by the external service provider. There is no problem if these incompatibilities are explicit because they can be easily detected and corrected—as it happens with signature incompatibilities, for example. These situations can be treated with the use of adaptors, wrappers, or any kind of adaptation techniques.

The major problem appears in the cases of implicit assumptions on the interaction models and mechanisms followed by clients and servers. Normally, these assumptions are implicitly made by software developers with previous knowledge about how the target platform(s) work. Whenever all the application is generated from the initial PIMs using a single platform technology, and therefore all parts follow the same interaction models and patterns, this problem does not happen. However, when we need to work with external entities, the interactions models of each party should be made explicit to be able to detect and resolve potential inconsistencies and conflicts at design level.

This work presents an approach for modeling the communication mechanisms between a Web application and its related external systems, that makes explicit both the programming abstractions through which the client and service provider view the communication, and certain implementation choices about the selected target platform that generally are implicit. This is specially relevant in those contexts in which several platform technologies may be simultaneously used.

In general, there is no standard way of describing implementation decisions such as concurrence, security or transaction aspects, in order to get computationally complete PIMS, i.e., PIMs that contain all the information required to produce real program code.

Several approaches that address this issue by identifying these concerns at different levels of abstraction have appeared recently. Almeida et al. [1, 2] introduce the *abstract platform* concept that defines characteristics that must have proper mappings onto the set of concrete target platforms that are considered for an MDA design process. Following a different approach that uses a UML profile, Witthawaskul and Johnson [22] define the *unit of work* concept which can be applied on a UML operation to support platform independent transaction modeling. Likewise, our work follows an approach based on marks (using a UML metamodel) that guide both the PIM to PSM transformation, and the PSM to the Implementation Model transformation too. They represent interaction model capabilities and services provided by potential target platforms abstracted and specified in a platform-independent way.

Another controversial issue is related to where implementation decisions are expected to appear: (i) directly in the PIM; (ii) in the target platform model, or; (iii) in the transformation model. The MDA community still struggles to deal with this issue, as a quick look at the discussion happening in the MDA mailing lists clearly reveals.

The structure of this document is as follows. After this introduction, Section 2 provides a brief description of the interaction styles supported by technologies like CORBA, Enterprise Java Beans, J2EE, Web Services or .NET. After that, Section 3 derives a UML metamodel based on the existing similarities found among the previous interaction models. Using this metamodel, Sections 4, 5 and 6 show how to apply it in a service-oriented scenario. Finally, Section 7 draws some conclusions and outlines some further research activities.

2 Interaction Models for Web Applications

Currently, Web applications need to interoperate with third party systems (external portlets, Web services or legacy applications) in a variety of ways—interaction models—which reflect the heterogeneity of applications built upon disparate implementation technologies such as J2EE, CORBA or .NET. Generally, each middleware technology has its own interaction model, although traditional client-server interaction patterns are likely to be common.

A Web application may require to communicate with a great variety of systems in different address spaces and running on heterogeneous platforms—which use different communication abstractions and interaction models. Here we will briefly describe the interaction styles of the most commonly used technologies, which are able to connect applications implemented using heterogeneous technologies.

2.1 Calling Service Provider for a Web Application

Currently, three main technologies support communication between modules of disparate systems, hiding platform and language specific details: CORBA, Enterprise Java Beans, J2EE and Web Services and .NET.

CORBA Service Provider. To request a CORBA service provider, the client may follow two approaches [8, 16, 20]: (i) a static invocation method or (ii) a dynamic invocation method. For the former, the client has to acquire at compile-time an object reference to the CORBA object. Then, this reference is used to initialize a proxy object that represents the remote object in the client's address space. For generating the proxy implementation, an IDL specification of the CORBA object is required and compiled into the client program. IDL specifications can define both synchronous (request/reply) operations and asynchronous (one-way) messages.

For dynamic invocations there is no information about the types and interface specifications of the required CORBA service. The client can look this information up by querying an Interface Repository (a service that provides IDL definitions at run-time). In consequence, a client request consists of operations for setting the name and parameters of the request and retrieving the returned values or an exception at run-time. Once the client has acquired a valid remote object reference to the CORBA server object, it can call the server object's methods as if the server object resided in the client's address space. The mapping of the object name to its implementation is handled by the Implementation Repository.

Enterprise Java Beans/J2EE Provider. For a client to call a business method, it needs to go via an EJB object (a generated Java class based on the Component Interface). This means that a client never accesses an enterprise bean directly [4, 21]. In this regard, the client has in first place to call a factory object (which is the EJB Home Object) to either locate an existing EJB object or create a new EJB object. Once it is generated during compile or deployment time, EJB objects act as bridges between the client and the bean instances [19].

There are several types of EJBs: *session beans, entity beans and message-driven beans*. The two former kinds of beans provide their interfaces to allow remote clients to invoke them. However, message-driven beans do no make public their interfaces. On the contrary, a message-driven bean listens for messages that are sent using *Java Message Service* and processes them anonymously (asynchronous invocations) [10].

An XML file describes how an Enterprise Java Bean should be assembled and deployed, its name, and other external dependencies of the bean.

Java/RMI. This mechanism is tied to the Java programming language and virtual machines [9]. RMI allows to invoke operations on Java objects. The client should contact first an RMI registry, and request the name of the service. RMI URLs identify services, including the hostname on which the service is located, and the logical name of the service. Then, the registry will point the client to the direction of the service it wants to call. The mapping of Object Name to its Implementation is handled by the RMI Registry.

RMI generates proxies and stubs from Java interface definitions. Furthermore, RMI uses Java's capabilities for dynamic linking to load the classes of parameters or returned objects over the network, allowing clients or servers to receive objects of classes unknown at compile time.

Web Services & .NET Provider. In order for a client to be able to consume a Web service, the client should know where the Web service resides and how to invoke its methods (that is, how to serialize the call to the Web service and how to deserialize received messages from the Web service). This information is provided by the WSDL specification of the Web service [23]—an XML document that specifies the data types of the messages, the protocols that are accepted, the Web service's endpoint, and the bindings.

Since notions involved in creating the SOAP message to send to the Web service, making the actual HTTP request, deserializing the HTTP response, etc. could be complex, they are abstracted using a proxy class. A proxy class is a class that encapsulates the complexity of calling a Web service and exposes this complexity through a simplified interface [12]. From the client application's perspective, the Web service is simply a local component—the client doesn't have to worry about the specifics of how to serialize a SOAP message, or how to make an HTTP request.

3 Modeling interactions

3.1 Interactions

The basic interaction model works according to the three-step process shown in Figure 1, being different interaction models supported by combinations of this configuration—mainly combinations of the second and third steps.

Step 1. A service provider publishes a description of their services in a publicly accessible registry.

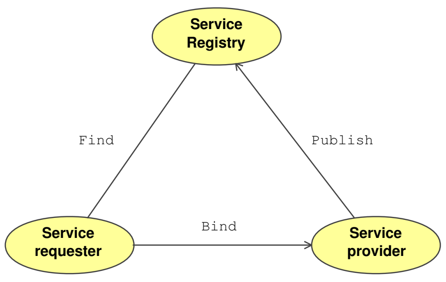


Figure 1. Basic Interaction Model

Step 2. A service requestor discovers those services by querying the registry and binds to the selected service. (Note that we will call the service requester a *client*)

Step 3. Client interacts with service provider.

According to Figure 1, an interaction between two endpoints can be defined in terms of:

- The set of messages accepted by the service provider (*Provided Interface*)
- The set of messages required by the client (*Required Interface*)
- A protocol that defines the partial order between the exchanged messages.
- The programming abstractions through which the client and server view the protocol (*the client-side and server-side programming interfaces*). This is important, because these programming abstractions encapsulate the agreement between both parties on: the data format, the mechanism for transforming and reconstructing object state into this format, the transport protocol, etc.

Most approaches focus just on the first three points. However, the fourth is not explicitly stated or modeled anywhere; instead, it is usually implicitly assumed by both the client and the server, and therefore hard-wired into their models, transformation rules, and code. This is neither flexible, nor provides the platform-independence required in a true MDA approach. Besides, these assumptions are usually separately made, which may cause contradictory assumptions. Thus, being able to express this kind of information—particulary the last point—in a Platform-Independent way is a step forward to achieve abstracts models established in more details that allow code-generation MDA tools to obtain really implementations.

3.2 Identifying Model Elements and their Relationships

UML Base Element	Stereotype
Port	≪ServerPort≫
Port	≪ClientPort≫
Port	≪StubClient≫
Port	≪ProxyClient≫
Port	≪DynamicClient≫
Interface	≪InterfaceSignature≫
Interface	≪ProvidedInterface≫
Interface	≪RequiredInterface≫
Assembly Adaptor	≪Interaction≫

Table 1. Summary of the stereotypes used

In our proposal we have tried to use as much as possible existing UML elements, in particular UML 2.0 elements because they provide some useful architectural concepts and mechanisms for our purposes. Table 1 shows a summary of the profile we have defined for representing these concepts. In particular, we consider each system as a UML 2.0 *Component*, which represents "a modular part of a system that encapsulates its contents, designs as well as implementations features, without losing the ability to describe deployment information and being replaceable within its environment" [17].

Component interactions are carried out through a layer of abstraction that allows clients to instantiate and access to the methods of the external services provider. In this sense, we can define one or more *Ports* through which a *component* invokes and receives method calls.

Since each endpoint can act as either a *provider* or a *client* in each of the Web interactions in which it plays a role, we have modeled causality by a *Port* stereotyped as *ClientPort* or *ServerPort*.

The interaction between a *ServerPort* and a *ClientPort* falls into one of the following categories:

- Synchronous invocation. The *ClientPort* invokes a remote procedure and blocks until a response or an exception is received from the *ServerPort*.

- Asynchronous invocation. The *ClientPort* invokes a remote procedure and continues processing without waiting for a return, although the returned value will be received in any moment.

- One-way invocation. The *ClientPort* invokes a remote procedure but does not block or wait to receive a return since it will not receive a return value.

In a first approach we will consider that each *Port* is associated with only one *Interface*. More precisely, a *ClientPort* is associated with a *RequiredInterface* and a *ServerPort* is associated with a *ProvidedInterface*. A *ProvidedInterface* specifies public operations that are remotely available. On the other hand, *RequiredInterfaces* complement *ProvidedInterface* and describe the features that make up a system that depends on in order to implement its functionality.

Similar to *Interfaces*, *Ports* describe how a *System* interacts with its environment, but is different in that *Interfaces* contain just syntactic information about methods provided by a *System* and *Ports* encapsulate the required business logic that allows a *Requirer* to interact with a *Provider*, tying that business logic with a concrete "implementation choice". Note that in many cases some implementation choices will only be supported by certain target platforms.

Each *Port* has associated a *Protocol* that defines the partial order in which the objects expect their methods to be called, and the order in which they should invoke other object's methods. *Port's Protocols* show a global perspective over its constituent external applications protocol descriptions. For simplicity we have supposed that each *Port* is associated with only one *Interface*, and hence *Port's protocols* will be coincide with *Interface's protocols*.

ServerPort's Protocol can be given inside text files as BPEL4WS or WS-CDL specifications, for example. On the other hand, *ClientPorts* interfaces can be augmented with behavioral descriptions based on *protocol state machines* that define usage constraints among features of the associated interface.

Many aspects of the *ClientPort* are determined by the external system the client connects to. In consequence, the *ClientPort* can be classified into three main categories based on the third party system that they can interact with:

(i) *Stub Clients* are never required to be downloaded or distributed to clients and they are specific for a certain protocol, transport option and server requirer (accorded at compile time). The client must obtain a reference to the *Stub* before using it, which represents an instance of the server provider. In order to obtain it, both the remote interface and its implementation have to be available so the client relies on an implementation-specific class.

(ii) *Proxy Clients*, as *Stub Clients*, refer to static invocation of server provider methods. They are not portable across implementations either—in this case, the code for the *Proxy Client* is created during runtime, but the reference to the interface specification of the external provider is obtained at compile-time.

(iii) *Dynamic Clients* can access a service discovering its interface description dynamically. In the same way, they can invoke server provider methods at runtime. This implies an extra work at runtime to fetch and process the server interface.

At this point, a benefit of using *Ports* is that the constraints and requirements on the communications between applications can be modeled without forcing software developers to take into account the platform specific notions in their designs. In this way, the designs can be reused to be run on different platforms (hence following the platform-independence philosophy dictated by MDA).

The kind of *Client Port* to be used is important, and strongly influences the kinds of client-side artifacts that need to be generated at development-time. Both *Stub* and *proxy* clients require the complete interface specification of the external services. That is, the client does not need to discover the required service but instead it has, at development-time, to know the external system's details (location, configuration file, WSDL or IDL URL, namespace, etc.). In contrast, *Dynamic Clients* must dynamically discover and invoke an external system without any prior knowledge of its details (signature of the remote procedure or the name of the service). For a *Dynamic Client*, there is no coupling between the service interface and the client. This makes the client code easy to modify if the external systems specifications change.

On the other hand, one of the most significant differences between *Stub Clients* and *Proxy Clients* is how external functionality is invoked. For the former, the client-side programming interface is embedded inside the client business logic. On the contrary, for *Proxy Client* and *Dynamic Client*, the client-side code is packaged apart from the client application.

Please note that the selection of these *Ports* only affects the client side. From the server perspective, it only receives and returns messages which are identical for all client types.

3.3 Adaptors

In case there is a strong requirement of using an external service provider (e.g., for *Stub Clients*), the software designer can specify what should be done if the behavior/specification of both parties is incompatible. Since we have available the interface of the required external systems (*provided interfaces*), we can carry out static checking for comparing them and determining whether they fulfil our requirements (*required interfaces*).

If not, the designer can decide at designing-time to implement an intermediate business logic (*adaptors*) that conforms to a given interface or consume a required external system. *Adaptors* mediates between the *ClientPort* and *ServerPort* interactions, resolving *service provider* and *service requester* differences at interface and protocol levels.

4 Example: The Travel Agency

In order to illustrate the use of interaction patterns in the definition of Platform Independent and Platform Specific Models, let us consider a Travel Agency service that sells vacation packages to its customers. The packages include flights, hotel rooms, car rentals, and combinations of these. External service providers include transportation companies (airlines, hotels and car rentals) and financial organizations (credit companies and banks).

To book a vacation package, the customer will provide details about his preferred dates, destinations, and accommodation options to the Travel Agency System (TAS). Based on this information, the TAS will request its service providers for offers that fulfill the user's requirements, and then will present the list of offers to the customer. At this point, the customer may either select one of the offered packages, reject them all and quit, or refine his requirements and start the process again. If the customer selects one of the packages, the TAS will book the individual services to the corresponding transportation companies, and charge the customer.

The straightforward application of MDA to develop a system is based on the following steps:

Step 1 Create class diagram (PIM) describing object model.

Step 2 Mark PIM elements with stereotypes.

Step 3 Customize the marked PIM with annotations.

Step 4 Specify the target platform.

Step 5 Generate a PSM.

In general, the MDA software development process becomes an iterative model transformation process where each step transforms one (or more) PIM of the system at one level into one (or more) PSM at the next level until a final implementation model is reached (see Figure2). Here, an implementation model is just another PSM, which provides all the information needed to construct a system and to put it into operation).

Note that, we will call *technology platform* to the last platform (i.e., the one that provides the executable PSM, or *implementation*). The intermediate platforms that transform PIMs into PSMs that will be used as PIMs in the next step are considered as *abstract platforms*.

Given that an element of the PIM may be marked several times with marks that come from different metamodels, it will be transformed according to each of the mappings. The semantic of the resulting marked element is given by the gathered features through the MDA model transformation process.

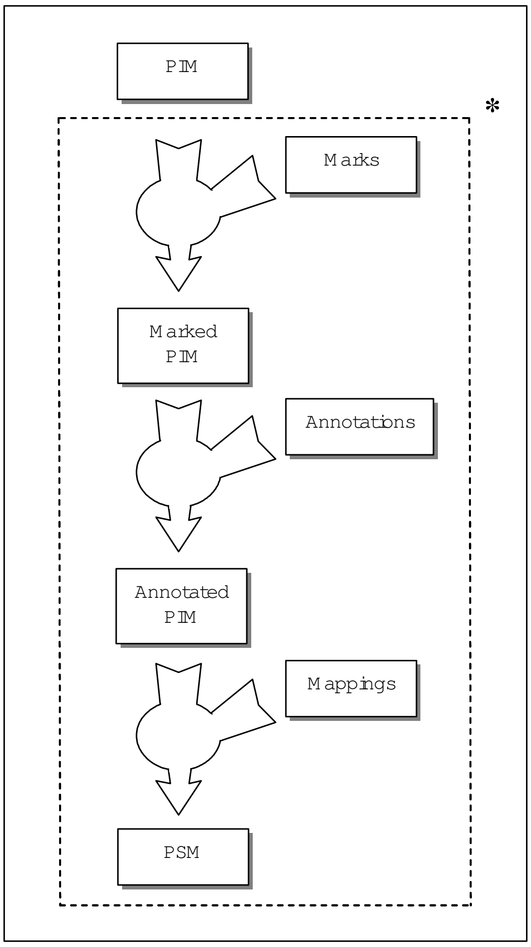


Figure 2. The PIM to PSM iterative process

In our approach we need to go though two main phases. Firstly, we need to identify the system scope and boundaries, i.e, which services will be provided by our system, and which ones will be externally required. The result of this phase is a high-level architectural view of the services and components of our global system. In the second phase, we need to determine the concrete platforms and communication mechanisms between our application and the external systems identified previously.

5 Identifying the scope and boundaries of our system

In our previous work [15], we presented a model-based framework that allows the high-level integration of Web applications with third party systems aligned with the MDA principles. It enables the manipulation of the external entities and systems as native elements of our models.

At design level, software developers are able to specify/mark: the system elements that require code generation; the system elements that will be remotely accessed using its provided interface specifications and implementations; the system elements that need to interact with others; and the system properties that are used for identifying them. All this is done in this first phase in a platform-independent manner, i.e., independently from the communication asbtractions and mechanisms used, and the platforms in which our system and the external services are implemented. These details will be added in the second phase.

In the first place we need to create the PIM of the system, which in our case is shown in Figure 3. It is focused just on the operation of the system, while hiding the rest of the details (software architecture, distribution, system boundaries, communication protocols, implementation platforms, etc.). This solution is specified in terms of UML packages and their interconnections in a platform independent way, where no implementation decisions have been explicitly specified (what greatly simplifies the application PIM making it reusable across different target platform environments).

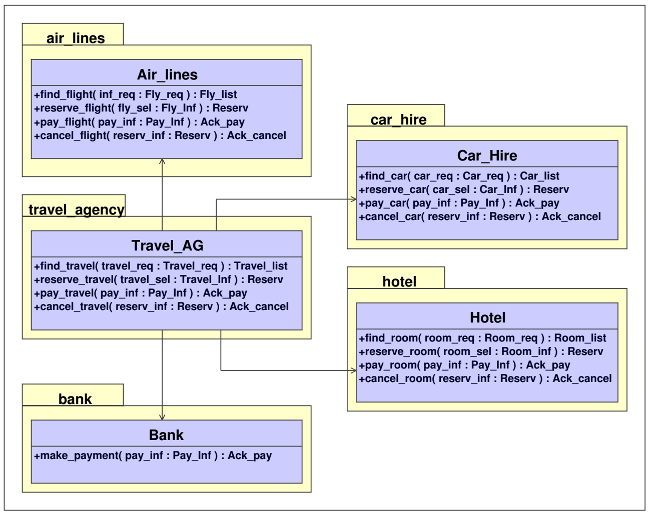

air_lines
Air_lines
+find_flight(inf_req : Fly_req) : Fly_list
+reserve_flight(fly_sel : Fly_Inf) : Reserv
+pay_flight(pay_inf : Pay_Inf) : Ack_pay
+cancel_flight(reserv_inf : Reserv) : Ack_cancel
car_hire
Car_Hire
+find_car(car_req : Car_req) : Car_list
+reserve_car(car_sel : Car_Inf) : Reserv
+pay_car(pay_inf : Pay_Inf) : Ack_pay
+cancel_car(reserv_inf : Reserv) : Ack_cancel
travel_agency
Travel_AG
+find_travel(travel_req : Travel_req) : Travel_list
+reserve_travel(travel_sel : Travel_Inf) : Reserv
+pay_travel(pay_inf : Pay_Inf) : Ack_pay
+cancel_travel(reserv_inf : Reserv) : Ack_cancel
hotel
Hotel
+find_room(room_req : Room_req) : Room_list
+reserve_room(room_sel : Room_inf) : Reserv
+pay_room(pay_inf : Pay_Inf) : Ack_pay
+cancel_room(reserv_inf : Reserv) : Ack_cancel
bank
Bank
+make_payment(pay_inf : Pay_Inf) : Ack_pay

Figure 3. The TAS PIM

As previously mentioned, any integration to legacy data and services may require that the interfaces to those elements are also modeled. The kind of information that is available from them will allow us to check whether they match our requirements or not, as described by the system model [14]. More precisely, this information should be able to allow us to:

(a) model the component or legacy system (e.g., by describing its structure, behavior, and choreography);

(b) check whether it matches the system requirements (this is also known as the *gap analysis* problem [7]);

(c) evaluate the changes and adaptation effort required to make it match the system requirements (i.e., evaluate the *distance* between the models of the "required" and the "actual" services, see e.g., [13]); and

(d) ideally, provide the specification of an adaptor that resolves these possible mismatches and differences (see e.g., [6]).

Although the integration of third party systems with a Web application should be address at three levels of abstractions (e.g., at presentation, business process and data level) [15], for the sake of simplicity in this paper we will only consider the process level.

Once the high-level PIM is identified, we need to identify the system scope and boundaries, and then build a model of the system with this information. That target model (PSM) will be built by transforming the original PIM using marks. To identify the elements in the TAS PIM that should be transformed in a particular way, we will use the stereotypes ≪ExternalSystem≫ and ≪ExternalAssociation≫. An ≪ExternalSystem≫ defines any other external system interacting with the system under consideration. In the same way, an ≪ExternalAssociation≫ defines an interaction between the system under deployment and an ≪ExternalSystem≫.

Implicitly, each type of model element in the PIM is only suitable for certain marks, which indicate what type of model element will be generated in the PSM.

Marks are not a part of the platform independent model although they appear on the marked PIM (see Figure 4).

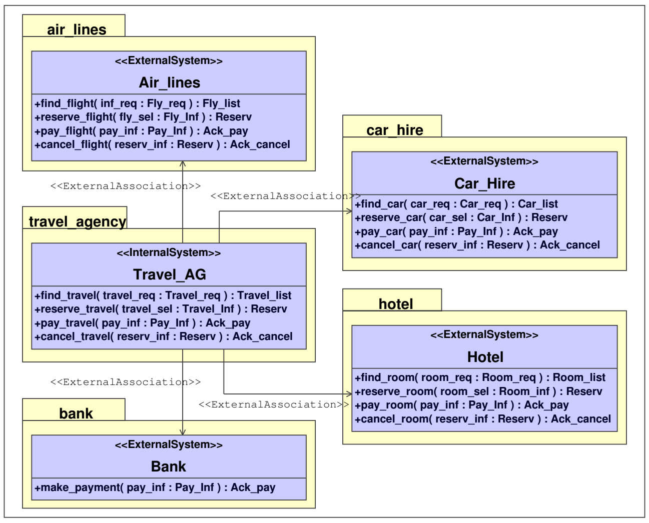

air_lines
<<ExternalSystem>>
Air_lines
+find_flight(inf_req : Fly_req) : Fly_list
+reserve_flight(fly_sel : Fly_Inf) : Reserv
+pay_flight(pay_inf : Pay_Inf) : Ack_pay
+cancel_flight(reserv_inf : Reserv) : Ack_cancel
car_hire
<<ExternalSystem>>
Car_Hire
+find_car(car_req : Car_req) : Car_list
+reserve_car(car_sel : Car_Inf) : Reserv
+pay_car(pay_inf : Pay_Inf) : Ack_pay
+cancel_car(reserv_inf : Reserv) : Ack_cancel
<<ExternalAssociation>>
<<ExternalAssociation>>
travel_agency
<<InternalSystem>>
Travel_AG
+find_travel(travel_req : Travel_req) : Travel_list
+reserve_travel(travel_sel : Travel_Inf) : Reserv
+pay_travel(pay_inf : Pay_Inf) : Ack_pay
+cancel_travel(reserv_inf : Reserv) : Ack_cancel
hotel
<<ExternalSystem>>
Hotel
+find_room(room_req : Room_req) : Room_list
+reserve_room(room_sel : Room_inf) : Reserv
+pay_room(pay_inf : Pay_Inf) : Ack_pay
+cancel_room(reserv_inf : Reserv) : Ack_cancel
<<ExternalAssociation>>
<<ExternalAssociation>>
bank
<<ExternalSystem>>
Bank
+make_payment(pay_inf : Pay_Inf) : Ack_pay

Figure 4. The marked TAS PIM

Note that the marked PIM is, by definition technology independent. In consequence, the prefix "External" used by the stereotypes ≪ExternalSystem≫ and ≪ExternalAssociation≫ in Figure 4 does not imply any implementation decisions. Instead, it is only used to limit the system scope that has to be development.

Once we have the marked PIM, we need to transform it into a PSM that can be translated into a target implementation code. As "platform" we will use here the UML 2.0 constructs and infrastructure for describing software architectures, because what we want to build in this phase is the software architectural description (i.e., model) of the system. This transformation will be guided by the following mapping rules:

- **Packages transformation.** Each UML package is mapped to a UML ≪Component≫ initialized with the same of its corresponding UML package.

- **Classes transformation.** The UML class stereotyped as ≪InternalSystem≫ or ≪ExternalSystem≫ is mapped to a UML ≪Class≫ holding the same characteristics as its original (name, attributes and operations).

- **Associations transformation.** For each UML association stereotyped as ≪ExternalAssociation≫ two component ports will be generated, each one as Association ends of that relationship. Ports will be associated to the UML ≪Component≫ derived in previous step. Its behavior is defined in terms of an interface associated with that port, which specifies the nature of the interactions that may occur over that port. Thus, the port interface's name is given the value of the UML class name from which it derives and its operations correspond to its UML class operations.

- **Association's ends transformation.** For the endpoint of an ≪ExternalAssociation≫ stereotyped as ≪InternalSystem≫ a usage dependency from the port to the interface is generated, showing how the ≪InternalSystem≫ provide a set of services.

  For the endpoint of an ≪ExternalAssociation≫ stereotyped as ≪ExternalSystem≫ an implementation dependency from the port to the interface is generated, showing how the ≪ExternalSystem≫ require a set of services.

- Finally, an assembly connector is defined from a required Interface to a provided Interface.

Applying these mapping rules on the PIM in Figure 4, we obtain the PSM shown in Figure 5.

As previously mentioned, the MDA software development process is an iterative model transformation process

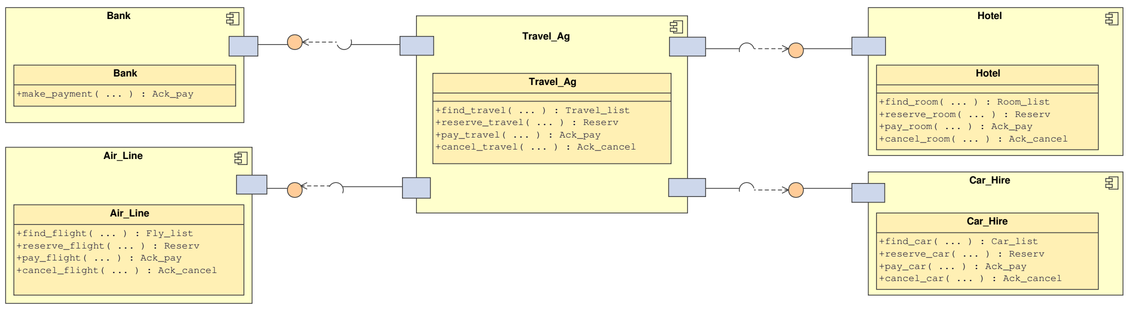

Bank
Bank
+make_payment(...) : Ack_pay
Air_Line
Air_Line
+find_flight(...) : Fly_list
+reserve_flight(...) : Reserv
+pay_flight(...) : Ack_pay
+cancel_flight(...) : Ack_cancel
Travel_Ag
Travel_Ag
+find_travel(...) : Travel_list
+reserve_travel(...) : Reserv
+pay_travel(...) : Ack_pay
+cancel_travel(...) : Ack_cancel
Hotel
Hotel
+find_room(...) : Room_list
+reserve_room(...) : Reserv
+pay_room(...) : Ack_pay
+cancel_room(...) : Ack_cancel
Car_Hire
Car_Hire
+find_car(...) : Car_list
+reserve_car(...) : Reserv
+pay_car(...) : Ack_pay
+cancel_car(...) : Ack_cancel

Figure 5. The PSM after applying the MDA transformation

whereby a PIM is transformed into a PSM, which in turn becomes the PIM for the next transformation—until a final PSM (the system *implementation*) is reached. What counts as a platform depends on the level of abstraction, and the kind of system being developed.

6 A Platform Specific Interaction-Model

Once we have the (UML 2.0) architectural description of the system, that identifies its scope and interactions with external services, the next phase focuses on the specification of such external interactions using the particular platforms and communication mechanisms of the required services. By adopting an MDA transformation process based on marks and annotations, we have to define the marks and transformations required.

Basically the information that the transformation process has to generate from the marked PIM is: the communication mechanisms between the *Components*; how the communications will be carried out; and the information that describes the architecture of the Web application. Therefore, the model shown in Figure 5 have to be marked again to specify that information.

Once we have applied previous transformation rules on the PIM, the resulting PSM is also platform-independent. We will mark them with decisions which are considered and taken in the context of a specific implementation design based on the concepts discussed in Section 3:

- Ports that specify services provided by external entities are stereotyped as ≪ServerPorts≫
- Ports that specify required services are stereotyped as ≪ClientPorts≫.
- Finally, assembly adaptors connecting interfaces have been stereotyped as ≪Interactions≫.

The resulting model is shown in Figure 6.

Now it is the time to include information about the technologies used to interact with the external services.

In the particular case of the Travel Agency System, we are going to make use of external service providers which include transportation companies (airlines, hotels and car rentals) and financial organizations (credit companies and banks). For illustration purposes we have selected different technologies for each external service. More precisely:

- A CORBA implementation of the Hotel Service. As previously mentioned, to participate in an interaction with a CORBA server application, the client (that is, our Travel Agency Service) must be able to get an object reference for a CORBA object and invoke operations on the object. To accomplish this, the client need information about references to the environmental objects that provide services for the CORBA application we plan to use and the IDL specification for implementing a stub-style invocation. Figure 7 shows how this information is specified using notes associated to its corresponding stereotypes.
- Another CORBA implementation of the CarHire Service. In this case, we plan to implement a dynamic interaction pattern so the IDL file will be looked up into an Interface Repository where it must be stored. In that sense, no IDL file has to be provided by the external server provider.

  The exact steps taken to access the Interface Repository depend on whether the client is seeking information about a specific object, or browsing the Interface Repository to find an interface. In both cases, before a dynamic client can browse the Interface Repository, it needs to obtain the object reference of the Interface Repository to start the search. Once the client has the object reference, it can navigate the Interface Repository, starting at the root.
- The two others external services are supposed to be available as external Web services. Their respec-

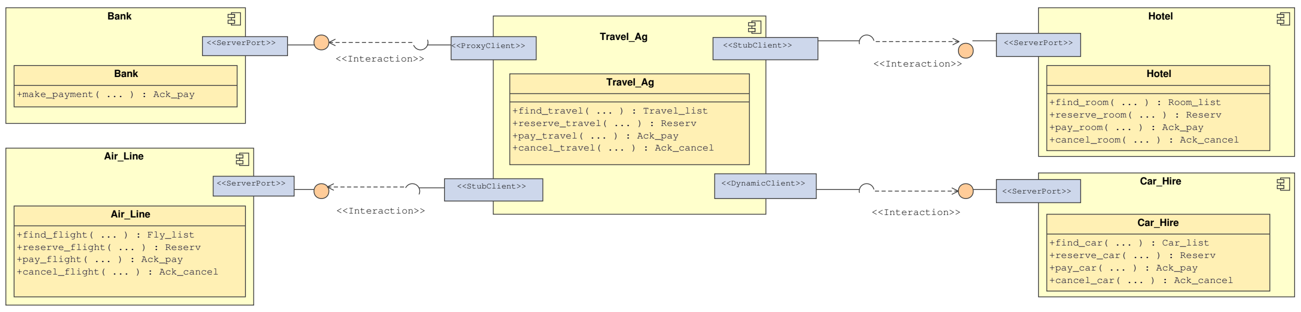

Bank
<<ServerPort>>
Bank
+make_payment(...) : Ack_pay
<<Interaction>>
<<ProxyClient>>
Travel_Ag
<<StubClient>>
Travel_Ag
+find_travel(...) : Travel_list
+reserve_travel(...) : Reserv
+pay_travel(...) : Ack_pay
+cancel_travel(...) : Ack_cancel
<<Interaction>>
<<ServerPort>>
Hotel
Hotel
+find_room(...) : Room_list
+reserve_room(...) : Reserv
+pay_room(...) : Ack_pay
+cancel_room(...) : Ack_cancel
Air_Line
<<ServerPort>>
Air_Line
+find_flight(...) : Fly_list
+reserve_flight(...) : Reserv
+pay_flight(...) : Ack_pay
+cancel_flight(...) : Ack_cancel
<<Interaction>>
<<StubClient>>
<<DynamicClient>>
<<Interaction>>
<<ServerPort>>
Car_Hire
Car_Hire
+find_car(...) : Car_list
+reserve_car(...) : Reserv
+pay_car(...) : Ack_pay
+cancel_car(...) : Ack_cancel

Figure 6. Marked PIM

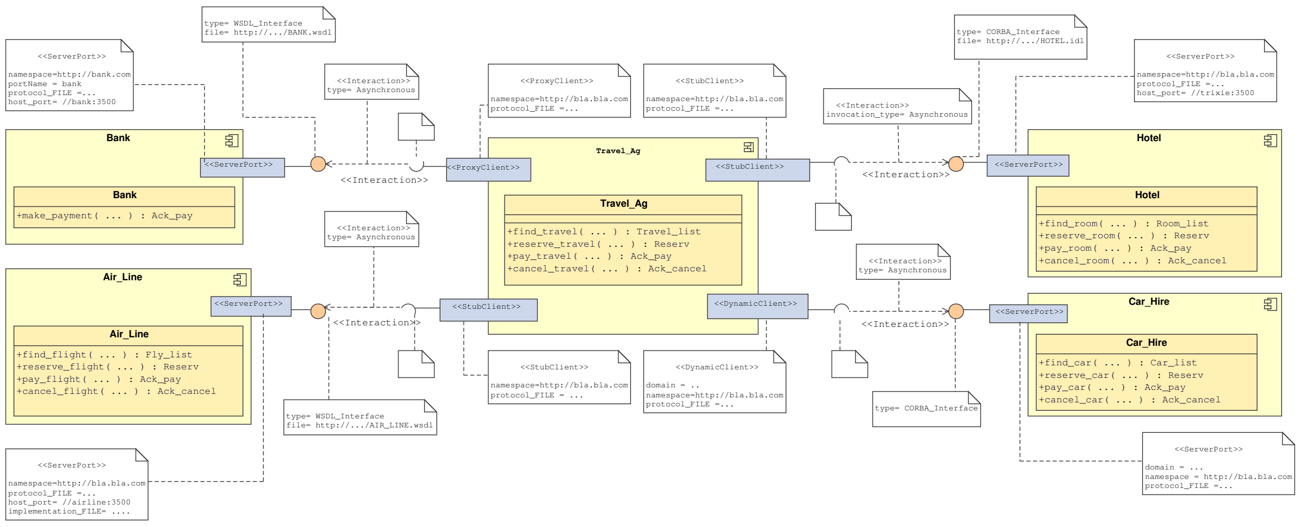

type= WSDL_Interface
file= http://.../BANK.wsdl
<<ServerPort>>
namespace=http://bank.com
portName = bank
protocol_FILE =...
host_port= //bank:3500
<<Interaction>>
type= Asynchronous
<<ProxyClient>>
namespace=http://bla.bla.com
protocol_FILE =...
<<StubClient>>
namespace=http://bla.bla.com
protocol_FILE =...
type= CORBA_Interface
file= http://.../HOTEL.idl
<<Interaction>>
invocation_type= Asynchronous
<<ServerPort>>
namespace=http://bla.bla.com
protocol_FILE =...
host_port= //trixie:3500
Bank
<<ServerPort>>
Bank
+make_payment(...) : Ack_pay
<<Interaction>>
<<ProxyClient>>
Travel_Ag
<<StubClient>>
Travel_Ag
+find_travel(...) : Travel_list
+reserve_travel(...) : Reserv
+pay_travel(...) : Ack_pay
+cancel_travel(...) : Ack_cancel
<<Interaction>>
<<ServerPort>>
Hotel
Hotel
+find_room(...) : Room_list
+reserve_room(...) : Reserv
+pay_room(...) : Ack_pay
+cancel_room(...) : Ack_cancel
<<Interaction>>
type= Asynchronous
<<Interaction>>
type= Asynchronous
Air_Line
<<ServerPort>>
Air_Line
+find_flight(...) : Fly_list
+reserve_flight(...) : Reserv
+pay_flight(...) : Ack_pay
+cancel_flight(...) : Ack_cancel
<<Interaction>>
<<StubClient>>
<<DynamicClient>>
<<Interaction>>
<<ServerPort>>
Car_Hire
Car_Hire
+find_car(...) : Car_list
+reserve_car(...) : Reserv
+pay_car(...) : Ack_pay
+cancel_car(...) : Ack_cancel
<<StubClient>>
namespace=http://bla.bla.com
protocol_FILE = ...
<<DynamicClient>>
domain = ..
namespace=http://bla.bla.com
protocol_FILE =...
type= CORBA_Interface
type= WSDL_Interface
file= http://.../AIR_LINE.wsdl
<<ServerPort>>
namespace=http://bla.bla.com
protocol_FILE =...
host_port= //airline:3500
implementation_FILE= ....
<<ServerPort>>
domain = ...
namespace = http://bla.bla.com
protocol_FILE =...

Figure 7. The Annotated TAS PIM

tive WSDL interface descriptions are required as illustrated Figure 7. Additionally, the code for the interaction with the Airline Web service relies on an implementation-specific class since it use an stub-style. This means that its implementation should be also available.

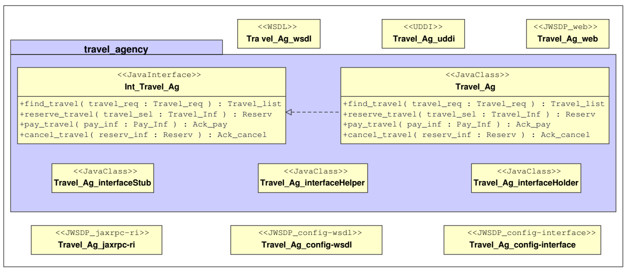
At this point, we also need to decide on the implementation technologies and platforms of our own system. Imagine that we decide to implement the Travel Agency using Java and Web Services technologies.

In this case we could use the transformation rules of any of the existing approaches for converting our marked and annotated PIM (in Fig. 7) to the corresponding PSM (shown in Fig. 8). For instance, we could follow the approach by Bezivin et al. [3], and then proceed according to the following steps:

1. Code the service endpoint interface and its implementation class. A service endpoint interface declares the methods that a remote client may invoke on the service. In this regard, each UML class is mapped to a ≪JavaClass≫ initialized with the same characteristics of its corresponding UML class. Based on it, the ≪JavaInterface≫ is also derived.

2. Build, generate, and package the files required by the service. In this sense, each UML class is also mapped to a ≪WSDL Specifications≫: ≪WSDL types≫, ≪WSDL operations≫, ≪WSDL bindings≫ and ≪WSDL services≫.

3. Deploy the service. Four deployment files are required: web.xml, jaxrpc-ri.xml, config-wsdl and config-interface. In this sense, the ≪JavaClass≫ is mapped to a ≪JWSDPweb.xml≫, ≪JWSDPjaxrpc-ri.xml≫, ≪JWSDPconfig-wsdl≫ and ≪JWSDPconfig-interface≫ files.



<<WSDL>>
Tra vel_Ag_wsdl
<<UDDI>>
Travel_Ag_uddi
<<JWSDP_web>>
Travel_Ag_web
travel_agency
<<JavaInterface>>
Int_Travel_Ag
+find_travel(travel_req : Travel_req) : Travel_list
+reserve_travel(travel_sel : Travel_Inf) : Reserv
+pay_travel(pay_inf : Pay_Inf) : Ack_pay
+cancel_travel(reserv_inf : Reserv) : Ack_cancel
<<JavaClass>>
Travel_Ag
+find_travel(travel_req : Travel_req) : Travel_list
+reserve_travel(travel_sel : Travel_Inf) : Reserv
+pay_travel(pay_inf : Pay_Inf) : Ack_pay
+cancel_travel(reserv_inf : Reserv) : Ack_cancel
<<JavaClass>>
Travel_Ag_interfaceStub
<<JavaClass>>
Travel_Ag_interfaceHelper
<<JavaClass>>
Travel_Ag_interfaceHolder
<<JWSDP_jaxrpc-ri>>
Travel_Ag_jaxrpc-ri
<<JWSDP_config-wsdl>>
Travel_Ag_config-wsdl
<<JWSDP_config-interface>>
Travel_Ag_config-interface

Figure 8. PSM

4. Generate client-side abstractions for consuming external services. For the CORBA services, we will add: the client stubs for each interface (interfaceStub.java), the CORBA helper class (interfaceHelper.java) and the CORBA holder class (interfaceHolder.java) that describe everything it is needed to use the client stub from the Java programming language. For the rest of the services, no more classes are generated. On the contrary, the code is embedded in the ≪JavaClass≫ implementation.

The PSM obtained, shown in Figure 8, includes all the details required to build the final implementation.

7 Conclusions and Future Works

In this paper we have discussed some of the (many) problems that may happen when integrating Web-based applications with external systems. In particular, we have concentrated on the interaction issues due to potential incompatibilities between clients and servers of different implementation platforms and middlewares. Our main contribution is to make such interaction models and mechanisms explicit, so incompatibilities can be detected, and bridges or adapters can be easily built. Besides, we have done it according to the MDA principles, encapsulating those interaction concepts and mechanisms in a platform-independent manner, and then providing transformation rules to the different implementations available of these concepts in most commonly used platforms and middelwares.

Now that the interaction issues are solved at this level, we plan to move forward, trying to address two other major issues. First, the (semi-)automatic derivation of adaptors in case incompatibilities are detected at this level. And second, move up one level of abstraction, and study how to model systems so potential mismatches that may happen at the behavioral semantic or contract (e.g., QoS) level can be detected.

References

[1] J. P. Almeida, R. Dijkman, M. van Sinderen, and L. F. Pires. On the notion of abstract platform in mda development. In *The 8th International IEEE Enterprise Distributed Object Computing Conference*, pages 253–263, Monterey, California, USA, Sept. 2004. IEEE Computer Society.

[2] J. P. Almeida, R. Dijkman, M. van Sinderen, and L. F. Pires. Platform-independent modeling in mda: Supporting abstract platforms. In *Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004)*, pages 217–231, June 2004.

[3] J. Bezivin, S. Hammoudi, D. Lopes, and F. Jouault. Applying MDA approach to B2B applications: A road map. *Workshop on Model Driven Development (WMDD 2004) at ECOOP 2004, Oslo, Norway, Springer-Verlag, LNCS*, 3344, 2004.

[4] D. Blevins. Overview of the enterprise javabeans component model. In *Component-Based Software Engineering: Putting the Pieces Together*, pages 589–606. Addison-Wesley, 2001.

[5] A. W. Brown. Model driven architecture: Principles and practice. *Software System Model*, 3:314–327, 2004.

[6] R. Bruni, H. Melgratti, and U. Montanari. Theoretical foundations for compensations in flow composition languages. *SIGPLAN Not.*, 40(1):209–220, 2005.

[7] J. Cheesman and J. Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing, Boston, MA, USA, 2000.

[8] M. Henning and S. Vinoski. *Advanced CORBA Programming with C++*. Addison-Wesley, 1999.

[9] JavaRMI. *Remote Method Invocation*. Sun Microsystems, 2004. http://java.sun.com /j2se /1.4.2 /docs /guide /rmi /spec /rmiTOC.html.

[10] Juan. *bla bla*, volume 3. p, p edition, p 2005.

[11] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Apr. 2003.

[12] Microsoft Corporation. *.NET Web Page*, 2004. http://www.microsoft.com/net/.

[13] R. Mili, J. Desharnais, M. Frappier, and A. Mili. Semantic distance between specifications. *Theoretical Comput. Sci.*, 247:257–276, Sept. 2000.

[14] N. Moreno and A. Vallecillo. What to we do with re-use in MDA? *Second European Workshop on Model Driven Architecture (EWMDA-2)*, Sept. 2004. Canterbury, Kent.

[15] N. Moreno and A. Vallecillo. A model-based approach for integrating third party systems with web applications. *Fifth International Conference on Web Engineering (ICWE2005)*, July 2005. Sydney, Australia.

[16] Object Management Group. *CORBA 3.0 - IDL Syntax and Semantics Chapter*, 2002. http://www.omg.org/docs/formal/02-06-39.pdf.

[17] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. http://www.omg.org/cgi-bin/doc?ptc/03-08-02.pdf.

[18] OMG. *Model Driven Architecture. A Technical Perspective*. Object Management Group, Jan. 2001. OMG document ab/2001-01-01.

[19] OpenEnterpriseX. *Just Enough Enterprise Java Beans Concepts*, 2004. http://www.OpenEnterpriseX.org.

[20] J. Siegel. *CORBA 3. Fundamentals and Programming*. John Wiley & Sons. OMG Press, 2000.

[21] Sun Microsystems. *The J2EE 1.4 Tutorial*, June 2004. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html.

[22] W. Witthawaskul and R. Johnson. Transaction support using unit of work modeling in the context of MDA. Available from http://weerasak.com/Unitf, Mar. 2005.

[23] *Web Services Description Language (WSDL) 1.1*, 2001. http://www.w3.org/TR/wsdl.