CLEI ELECTRONIC JOURNAL, VOLUME 8, NUMBER 1, PAPER 6, AUGUST 2005

Architecture of Component-based Information Systems over the J2EE Platform

Daniel Perovich, Leonardo Rodríguez, Andrés Vignaga
University of Uruguay, Engineering School, Computer Science Department, COAL Group
Julio Herrera y Reissig 565, 5th floor
Montevideo 11300, Uruguay
{perovich,lrodrigu,avignaga}@fing.edu.uy

Abstract

Component-based development can be addressed from two different fronts, or more precisely, from two different levels. One of them regards the technology used for system implementation, and the other is a previous and more abstract level, where the focus is set to the logical structure of the solution and where technological issues are not considered. Model Driven Architecture promotes such separation by distinguishing platform independent models from platform specific models. In alignment with this approach, this article proposes a mapping from the tiered and platform independent architecture for information systems resulting from the application of a widely known methodological approach, to the available constructs in the J2EE platform. This mapping allows the definition of transformations between platform independent models, resulting from the referred methodology where it is possible to abstractly reason about the solution, and platform specific models which are aligned with technological constructs and are directly implemented.

Keywords: Software architecture, Component-based development, Information systems, Java 2 Enterprise Edition, Enterprise Java Beans, Model Driven Architecture.

1 Introduction

Component-based development (CBD) is continuously evolving. Particularly, such evolution is experimented in two different worlds. In the one hand, there are studies on CBD at a conceptual or even methological level. In such studies, building on top of a concrete notion of component, a logical architecture of some specific kind of systems and the steps towards populating it with components are treated. On the other hand, there are technologies which provide platforms for component development and execution, such as Java 2 Enterprise Edition (J2EE) [14]. In that world, the notion of component is defined as an implementation construct. Despite of the fact that existing platforms share some commonalities, each ultimately imposes its particular approach, which in turn evolves with every new version of the technology. It can be argued that the first world is independent of the second world, since its concern is to express solutions in terms of an abstract notion (technology-independent) of components. This approach is widely known as "model driven". However, such models must be implemented using constructs provided by a concrete technology. For that reason, a connection between the two worlds becomes necessary.

It is possible to conceive a solution completely in the technological world, however it carries a number of drawbacks. For example, the solution is tightly coupled to a particular technology, or even worse, to a particular version of the technology. Reasoning about the solution is then obscured by technology dependent details such as terminology and approach. Furthermore, it is not sufficient for an architect to have an adequate knowledge of the technology, instead he or she must be an expert. A model driven approach allows for an abstract reasoning about the solution and constitutes an interesting approach for managing the complexities involved in component-based software development. Works such as [3, 21] follow that approach.

Efforts for bridging these two worlds are currently carried out. The Object Management Group (OMG) with its proposal of Model Driven Architecture (MDA) [10], looks for a separation between business logic and the technology of the underlying platform. In this way, platform independent applications built according to the MDA approach can be realized in different proprietary platforms. The concept of model is truly central in MDA. Particularly, two kinds of models are distinguished: Platform Independent Model (PIM) and Platform Specific Model (PSM). In MDA, a PIM generated for one system can be transformed into different PSMs. Then, a PSM can be directly implemented in the specific platform.

This work is aligned with the MDA approach. Starting from a model of a system constructed in a technology-independent fashion (i.e. a PIM) and applying a particular methodology, our main goal is to define a mapping to the constructs of a particular technology. This means that we look for all necessary elements for generating a platform specific model (i.e. transform a PIM into a PSM), instead of code generation techniques starting from a PIM. We focus on PIMs built following the approach in [3], which in turn is based on [4]. In that work, the steps for specifying a component and layered architecture of an information system are described. The target platform of the mapping is J2EE. Such mapping was marginally studied in [3]; in this work we perform a wider and deeper treatment of this issue.

The rest of the document is organized as follows. Section 2 reviews the basic notions of the methodological approach of [3] and introduces a PIM resulting from its application. In section 3, the correspondence between the different kind of components found in a PIM and the constructs of J2EE is presented. This constitutes a basis for the generation of PSMs for such platform. Section 4 presents the generated PSM from the PIM introduced in section 2 applying the concepts of section 3. Section 5 concludes.

2 Technology-Independent Architecture of Information Systems

In this section we review the fundamental ideas introduced in [3]. The architecture of an information system is specified in two levels of refinement. The first one, called System Architecture express the architectural style to be applied in the highest level of abstraction. As already mentioned, the architectural style used here is Layers. In the first part we enumerate the concrete layers to be used together with a description of the main responsibilities of the components in each of them. In the second part, based on the stated responsibilities we study the basic elements of a logical component architecture specification, which constitute our PIMs. An example of one of them is also shown.

2.1 System Architecture

Information systems can be organized according to the Layers architectural style [12]. This architectural style is organized hierarchically. Every layer provides services to their upper adjacent layer and acts as

a client to their lower adjacent layer. Services provided by higher layers have a high level of abstraction, and services provided by lower layers have a low level of abstraction. In what follows, the organization of an information system and the responsibilities of the components in each layer are presented. Layers are introduced in decreasing order of abstraction and match those presented in [3] with the addition of the *User Dialog UI* layer:

- User Interface Layer. It is responsible of presentation to the users. It contains forms, web pages, etc.
- User Dialog UI Layer. It handles UI logic and coordinates the presentation to users.
- User Dialog Layer. It handles the dialog's logic and maintain their state.
- System Services Layer. It provides operations that fulfill system requirements. It is organized in subsystems. It also contains adapters to external systems.
- Business Services Layer. Its components correspond to stable business types. It manages the persistent business information.
- Infrastructure Layer. It provides basic services such as security, transactionality, data caching, etc. It usually involves frameworks, APIs, database management systems, etc.

2.2 Logical Architecture

The system architecture presented in the previous section defines the general style of the application. The next step is to define the style to internally apply to every layer, as well as the elements that will populate them.

The internal organization of each layer varies. The user interface, if web-based, will follow a Client/Server style. The chosen methodology may affect the architectural style to follow, since by applying the methodology every layer is populated with elements which will constitute it. The example in Figure 1, treated in [11], was developed following the approach presented in [3, 21]. As a consequence, the architectural style for User Dialog, System Services and Business Services is *components*.

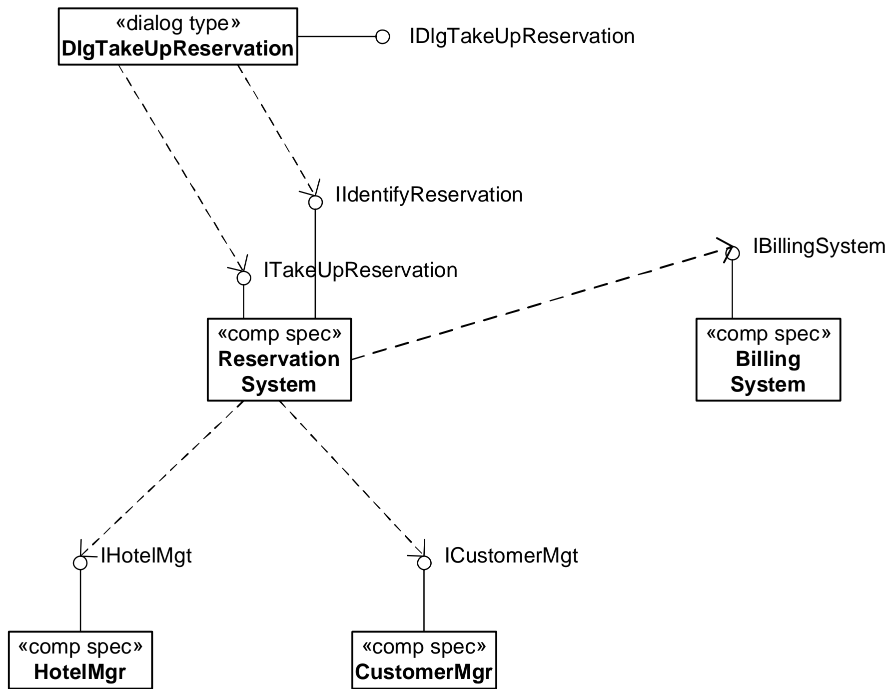


Figure 1: technology-independent logical architecture

Stereotype ≪comp spec≫ denotes a component specification. Each specification offers one or more interfaces, which are stereotyped with ≪interface type≫. An interface type has an associated information

model which is the basis for expressing the contracts for each of its operations. User dialogs are not considered in [3]. A user dialog, stereotyped with ≪dialog type≫, encapsulates the use case logic. Since it does not have an associated information model, its provided interface does not need to be specified by an interface type; a regular interface suffices for what needs to be specified.

3 Mapping to the J2EE Platform

In this section we discuss the details of the mapping between the elements of every layer of the system architecture and the constructs of the J2EE platform. With such a mapping it is possible to transform a PIM, such as that shown in the logical architecture, into a PSM for the J2EE platform. Since we used Enterprise JavaBeans (EJB) [15] in those layers whose style is components, we applied an UML profile for EJB [6] for their modeling. In what follows we tackle each layer in the system architecture, and examples will be based on the logical architecture already introduced.

3.1 User Interface

The User Interface layer was not included in the logical architecture. With the advances in Internet technologies, more and more companies are turning to web applications as they use Intranets. Considering this, we define the mapping using web technology.

This layer is usually organized according to the Client/Server style. At the client a browser runs. It presents to the final user web pages based on HTML and HTML Forms. By these means, information is gathered and communicated to the server.

At the server side a web server with dynamic web pages generation capabilities is required. To that end, JavaServer Pages (JSP) [16] is the technology used. JSP pages are built from a tag language and allows to define dynamically generated pages through templates. Such templates include HTML tags for static content and special JSP tags for dynamic content. For a complete actor-system interaction (i.e. a use case) more than one web page is usually necessary. A JSP page will be used for every stage of the interaction. Forms will be redirected (by means of the right setting of the `action` attribute) to an URL served by a servlet [17] located in the lower layer. In turn, such servlet is responsible of providing the page the necessary information for content generation. For this purpose the EJB pattern Data Transfer Object [9] (or one of its variants) is used. According to this pattern, information should be packaged in serializable classes, for which we use JavaBeans [13]. This will be the mechanism to interchange information through all layers.

3.2 User Dialog UI

The User Dialog UI layer uses a variant of the GRASP pattern Facade Controller [8]. Every page deliver their responses to this controller. A servlet is used for this purpose, which is responsible of receiving information from users (through HTTP requests) and invoking the corresponding user dialog (use-case controller). The servlet, through its session object, has an associated user dialog to which it will redirect all requests. At every request it sends the received information (encapsulated in a JavaBean), retrieves the result, and responds the user with a new JSP page which is built with the data received from the dialog.

This is an application of the Model-View-Controller pattern [2], where the JSP pages are the "View", the servlet is the "Controller", and the layer to be presented next are the "Model". An alternative modeling of these two layers would involve the use of web application frameworks, such as Struts [7], which is left as future work.

3.3 User Dialogs

A user dialog encapsulates use case logic. In that sense, it implements a state machine (derived from a use case specification) which is used for knowing which services to request from the lower layer and in which order, as well as the next page to be displayed. The servlet defined for the previous layer does not implement such state machine. We do not expect it to know the dialog's operation that correspond to a given request. To that end we apply the GoF [5] design pattern Command. We define an interface named `IDialog` whose only operation is `execute(in datain:Data):Result`. Type `Data` is the superclass of every data interchanged between every dialog and the higher layer. It is defined as a JavaBean. Furthermore, `Result` is also a JavaBean that encapsulates an instance of `Data` representing resulting data, and an identifier

of the state of the dialog after execute is completed. In the higher layer, the servlet uses this identifier for choosing the next JSP page, which is built with the data received in the `Result` instance. The mapping from the identifier to the JSP page can be defined by means of a configuration file (such as an XML file), so the servlet remains unaffected by changes in either the user dialogs or JSP pages. A user dialog is implemented using a Stateful Session Bean, applying the GoF design pattern State. Such EJB allows for maintaining the state of the interaction with the client (in our case, the state machine) along the use case.

In summary, a sample scenario of action between the three layers specified so far can be described as follows. A user provides information through the form presented in his or her browser. After the information is submitted, it is directed to the web server and received by the servlet. The servlet then invokes the execute operation of its associated dialog, passing an instance of the `Data` JavaBean which encapsulates the information submitted by the user. A dialog, depending on its state and the received information, issues a request to the system services layer. When the result is available, it chooses its next state and returns a `Result` instance which includes an instance of `Data` corresponding to the result and the identifier of the new state. In that way, the servlet sends to the web client the JSP page that corresponds to the received state identifier, which in turn is built using the information contained in the instance of `Data`.

It is important to take into account what kind of granularity is needed in terms of transactionality. Using container-managed transactions it is only possible to mark transactions at the operation level. This means that if transactions are managed by the container for the user dialog interface, then a transaction could only be related to one of its operations. However, we need an entire use case to be included into one transaction. For that reason bean-managed transactions should be used instead. In that case, when the first invocation to execute is received a new transaction is created. As we are using stateful session beans, the transaction will be preserved across subsequent invocations. The state machine implemented in the bean will be capable of signaling when the commit should be done, or rolling back when the use case fails. Figure 2 presents an internal view of `DlgTakeUpReservation` stateful session bean. Note that the Component Interface of the user dialog extends the `IDialog` interface mentioned before.

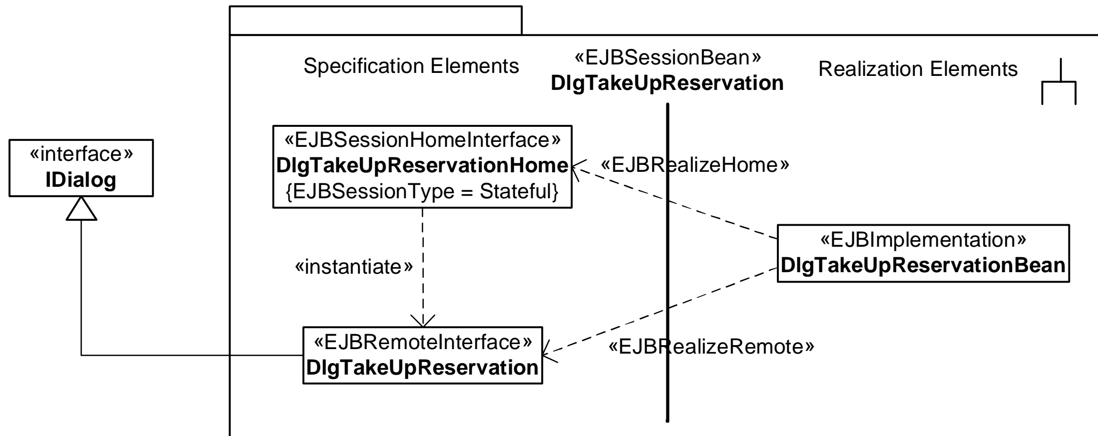


Figure 2: Internal view of `DlgTakeUpReservation` session bean

3.4 System Services

A business process rises a set of use cases. Following the methodology, a system interface is created for each detected use case. These use cases, originated from the same business process, are highly cohesive. For that reason, we create a unique system component that realizes all the system interfaces introduced by such business process.

In the EJB component model, a component may realize one and only one Component Interface. For a component to realize more than one interface, the Component Interface could be sub-interface of all interfaces offered by the component. Defining whether the Component Interface is local or remote is not a minor issue. Commonly, when the User Dialog layer is located in the same application server as the System Services layer, for performance reasons a local interface is preferable. If these layers should be distributed, a remote interface would be necessary. The same criteria applies when defining the kind of interfaces in lower layers.

In our case study we assume that User Dialog, System Services and Business Services Layers are located in the same application server, so only local interfaces will be used.

Such component acts as a facade within the business process, grouping all the operations involved. Components in this layer are Facade Controllers, again following GRASP. Session information concerning the dialog between the actors and the system is maintained in the User Dialog layer; components at this level do not need to maintain state. For that reason, components in this layer are implemented using Stateless Session Beans. Note that System Services layer follows an EJB pattern Session Facade [9] approach. It is important to note that every operation at this level generally need to be included in a transaction. This is specified declaring for every operation the `TX_REQUIRED` attribute. Figure 3 presents an internal view of the `ReservationSystem` stateless session bean.

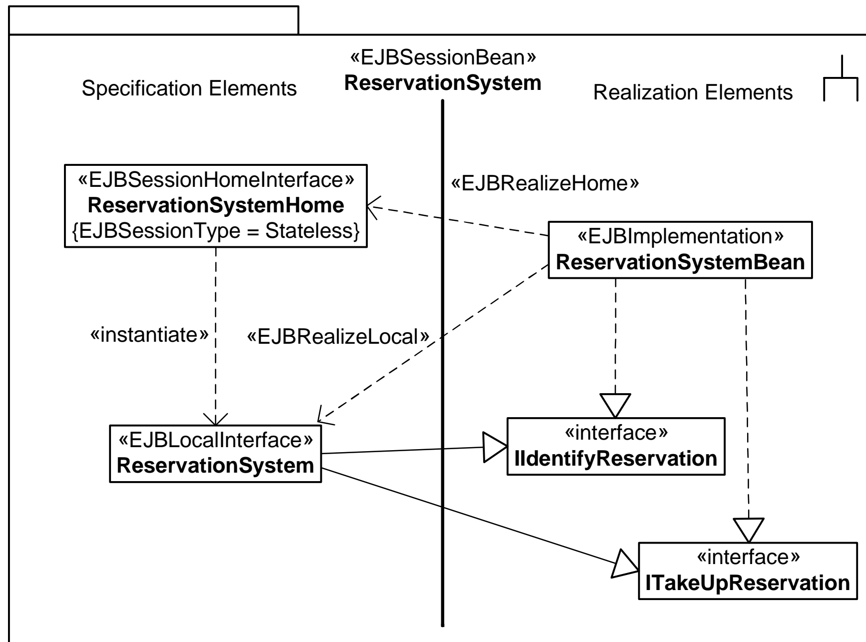


Figure 3: Internal view of `ReservationSystem` session bean

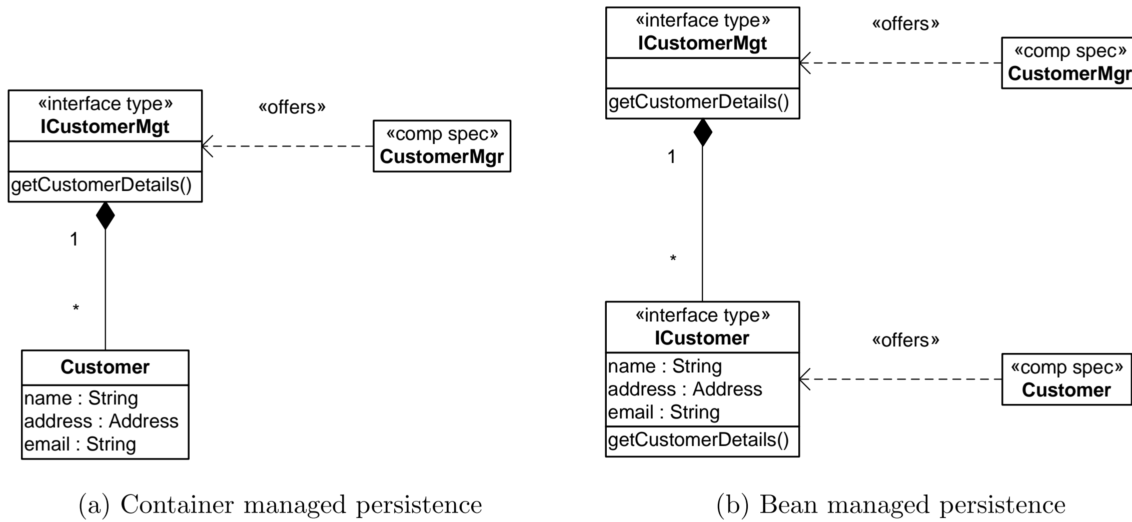
3.5 Business Services

Components in the Business Services layer were detected from the Business Type Model. The methodology suggests that a business interface should be created for every "core type" in the model in order to manage their instances (e.g. `ICustomerMgr`). For accessing and further processing such instances an identifier (e.g. `CustomerId`) should be provided to the manager. For example, an operation in `ICustomerMgr` such as `getCustomerDetails()` accepts the customer identifier as a parameter and returns the customer details.

A manager in EJB is also implemented using a Stateless Session Bean. Every component requiring services from a manager must request an instance of it. This approach is simple, applicable in many cases and supports interfaces as specified. Once it has been decided that a manager is a session bean, it is necessary to define the way the manager manipulates the information it manages and which may be stored in different data sources.

One possible approach is that session beans use Java DataBase Connectivity (JDBC) [18] for manipulating the information. One clear drawback of such approach is that the developer must explicitly implement the persistence of the application. This leads to an fuzzy separation of business types management and their persistence, besides known problems as SQL-Spaghetti. For minimizing such problems the J2EE pattern Data Access Object [1] can be applied, encapsulating the access to data sources. An interesting alternative is using Java Data Objects (JDO) [9, 20] as a persistence mechanism. It provides an automatic mapping from a Java object model into the relational model. In addition, it supports a number of interesting features such as transactions and security. Although this technology is not part of the J2EE platform it can be easily integrated, as explained in [9]. Further details about JDO are beyond the scope of this work. Both

approaches share the fact that a session bean acting as a manager is responsible of managing the persistence of its own information. Figure 4 (a) sketches this.



(a) Container managed persistence (b) Bean managed persistence

Figure 4: Two approaches for designing business services

Another alternative is to further apply the component model below the level of the managers, considering the instances of `Customer` as component instances (*component object*). In this approach the specification of some manager's features is simplified, since instances need to be dereferenced in less measure. Instance specific operations will be defined in an interface that realizes `Customer`'s interface type, that is `ICustomer`. An Entity Bean can be used here to implement `ICustomer` interface, where each instance of the bean represent a customer. This allows for the use of container-managed persistence for storing and retrieving the structures of a customer. In addition, since EJB 2.0 [15], the EJB Query Language (EJB-QL) can be used for querying for information independently of the query language specific to the data sources. Figure 4 (b) sketches this approach.

This approach allows for taking advantage of the benefits of an EJB container at the level of business entities (such as security, transactions, pooling, cache, fail-over, clustering, among others). Particularly, it promotes component portability, which would be affected if JDO is used (as it is not a J2EE technology) or even JDBC (if the database engine is changed). However, for some particular operations this approach could be inefficient, for example in cases where operations affect several entity beans. A thorough comparison of both worlds is well beyond the scope of this work. For further details on this matter we refer to [9]. The approach to be used in our case study is based on entity beans. Figure 5 and 6 show an internal view of the `CustomerMgr` session bean and `Customer` entity bean respectively.

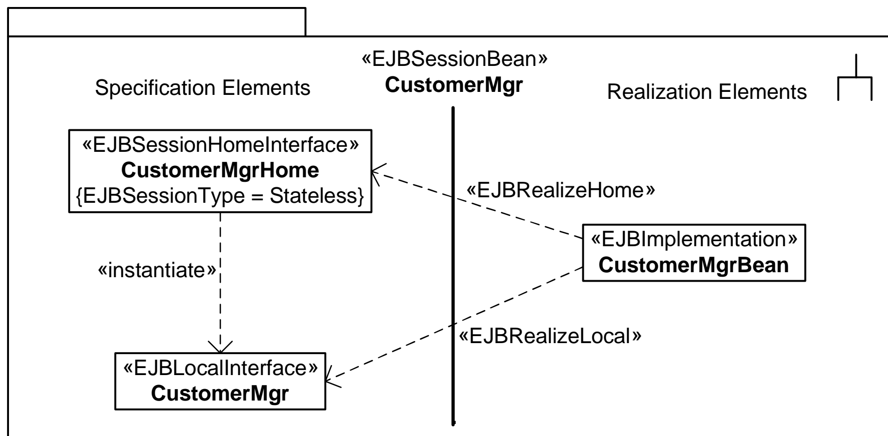


Figure 5: Internal view of `CustomerMgr` session bean

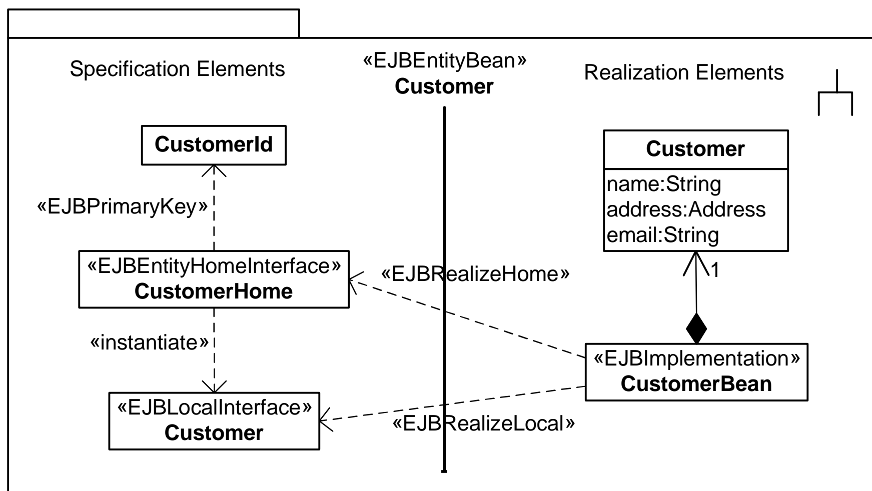


Figure 6: Internal view of `Customer` entity bean

As already mentioned, both entity and session beans in this layer export their services in this work through local interfaces. If necessary, session beans could use remote interfaces; however, in the case of entity beans, for reasons regarding performance it is extremely important the use of local interfaces.

3.6 Integration with Existing Systems

Integration with existing systems can be achieved using either APIs provided by an existing system or Enterprise Application Integration (EAI) software.

In our case study we use an existing Billing system. It is necessary to develop an adapter (applying the GoF pattern Adapter) which provides a bridge from our component environment to the external system. In terms of the mapping to the J2EE platform, we use J2EE Connector Architecture (JCA) [19] to access the Billing system through the Common Client Interface (CCI). By JCA, and assuming that an XA-Resource adapter is available, the external system can be included in transactions as needed.

A session bean is then included for supporting the necessary interface of the existing system. Such bean will redirect (probably adapting ingoing and outgoing data) requests to the external system. In this way we centralize in one component the use of JCA thus preserving the desired qualities of a component-based system. Figure 7 shows an internal view of the `BillingSystem` session bean.

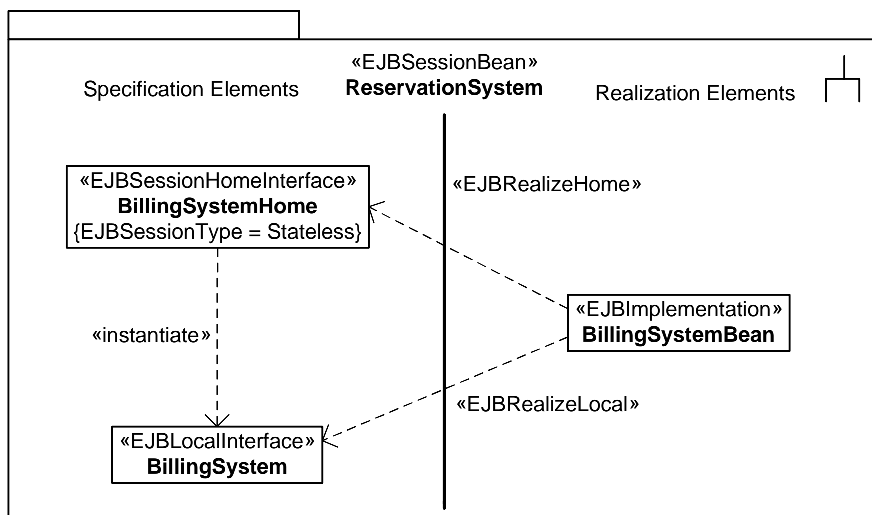


Figure 7: Internal view of `BillingSystem` session bean

4 Example of Platform Specific Model for J2EE

In this section we present the platform specific model (shown in Figure 8) corresponding to the platform independent model shown in Figure 1. This view unifies all the different mappings studied in section 3, but referring only to those layers populated with components, i.e. User Dialogs, System Services and Business Services.

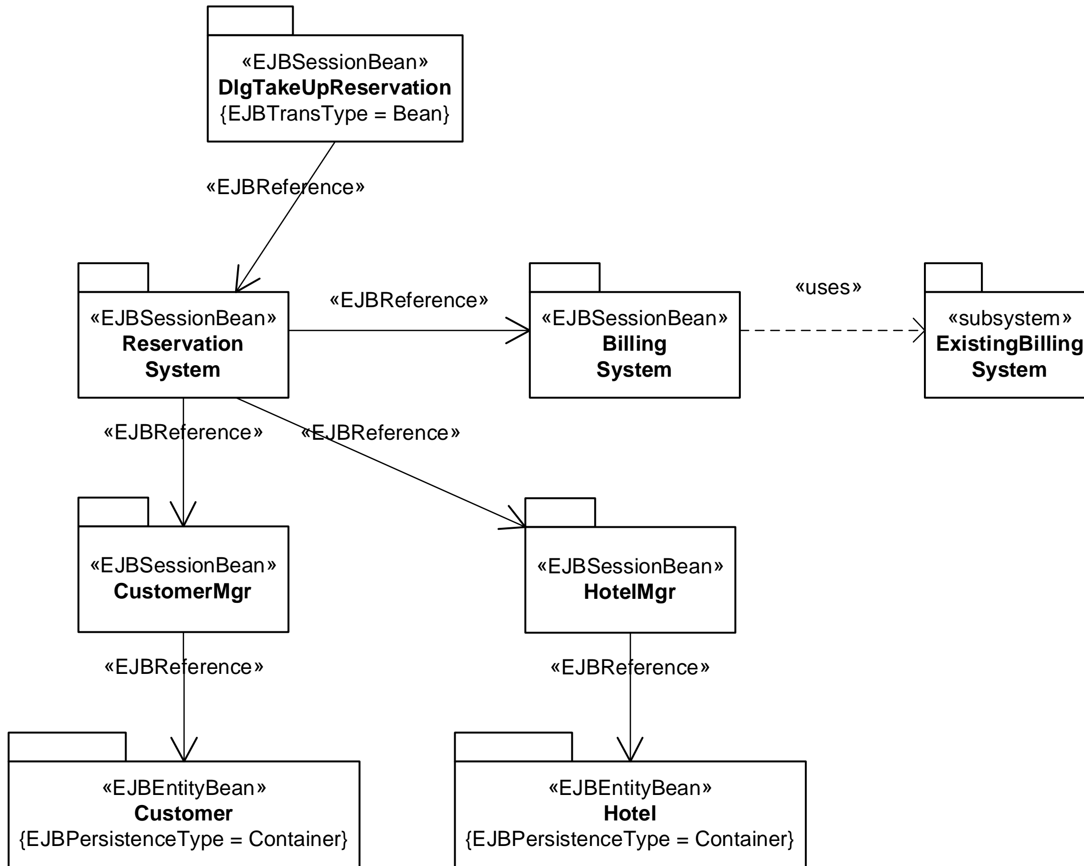


Figure 8: Logical architecture specific for J2EE

Along the use case treated in this work, it can be seen that models in every figure share some similarities in their structure. The fact that in Figure 8 two beans corresponding to a business component are present (e.g. component specification `CustomerMgr` correspond to the session bean `CustomerMgr` and entity bean `Customer`) is because we chose the entity bean approach for the Business Services layer. Applying an approach based on JDBC or JDO would lead to a different model.

5 Conclusions

The architecture of an information system is usually organized according to the architectural style Layers. The number of layers and their responsibilities depend on the methodological approach to be applied for the system development. In [3] an architecture for such systems is defined, and a proposal for a technology-independent specification of components as the building blocks of some of the layers is introduced. Such specification architecture for a particular system corresponds to a PIM in the MDA approach. In this work that proposed architecture for information systems was refined with the addition of a new layer, *User Dialog UI* layer, and a mapping from the elements in every layer of the system architecture to the constructs of the J2EE platform was defined. Such mapping allows a transformation to a specification architecture. The result of such transformation is expressed in specific terms of the platform and corresponds to a PSM in the MDA approach. The separation between PIMs and PSMs and the notion of transformation from one to the other

constitute the very essence of MDA. This work presents a study which make one of such transformations possible.

Further work includes the complete specification of the example presented in this work, generating the logical architecture specific to J2EE of the case study treated in [11]. A complete implementation of the case study could be realized, which in turn would allow for refining the mapping here presented. In addition, it is also of our interest to model the two top-most layers of the system architecture in terms of a web application framework, as well as the study of analogous mappings to other technologies, such as the .NET framework.

References

[1] Alur, D. and Crupi, J. and Malks, D. *Core J2EE Patterns: Best Practices and Design Strategies.* First Edition. Prentice-Hall, 2001.

[2] Buschmann, F. and Meunier, R. and Rohnert, H. and Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons, 1996.

[3] Cheesman, J. and Daniels, J. *UML Components: A Simple Process for Specifying Component-Based Software.* Addison-Wesley, 2001.

[4] D'Souza, D.F. and Wills, A.C. *Objects, Components and Framekorks with UML: The Catalysis Approach.* Addison-Wesley, 1998.

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

[6] Greenfield, J. *UML Profile for EJB.* Rational Software Corporation, Public Draft. Internet: `http://www.jeckle.de/files/UMLProfileForEJBPublicDraft.pdf`, 2001.

[7] Husted, T. and Dumoulin, C. and Franciscus, G. and Winterfeldt, D. and McClanahan, C.R. *Struts in Action: Building Web Applications with the Leading Java Framework.* Manning Publications Company, 2002.

[8] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* Second Edition. Prentice-Hall, 2001.

[9] Marinescu, F. *EJB Design Patterns. Advanced Patterns, Processes and Idioms.* First Edition. John Wiley & Sons, 2002.

[10] OMG. *Overview and Guide to OMG's architecture, MDA Guide Versin 1.0.* Object Management Group. Internet: `http://www.omg.org/docs/omg/03-05-01.pdf`, 2001.

[11] Perovich, D. and Vignaga, A. *SAD del Subsistema de Reservas del Sistema de Gestin Hotelera.* Thechnical Report RT03-15, InCo Pedeciba, Montevideo, Uruguay, 2003.

[12] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[13] Sun Microsystems. *JavaBeans Specification. Versin 1.0.1.* Internet: `http://java.sun.com/products/javabeans/docs/spec.html`, 1997.

[14] Sun Microsystems. *Java 2 Enterprise Edition Platform Specification. Versin 1.3.* Internet: `http://java. sun.com/j2ee/1.3/download.html`, 2001.

[15] Sun Microsystems. *Enterprise JavaBeans Specification. Versin 2.0.* Internet: `http://java.sun.com/products/ejb/docs.html`, 2002.

[16] Sun Microsystems. *JavaServer Pages Specification. Versin 1.2.* Internet: `http://java.sun.com/products/jsp/download/index.html`, 2001.

[17] Sun Microsystems. *Java Servlet Specification. Versin 2.3.* Internet: `http://java.sun.com/products/servlet/download.html`, 2001.

[18] Sun Microsystems. *Java DataBase Connectivity Specification. Versin 3.0.* Internet: `http://java.sun.com/ products/jdbc/download.html`, 2001.

[19] Sun Microsystems. *J2EE Connector Architecture. Versin 1.5.* Internet: `http://java.sun.com/j2ee/connector/download.html`, 2002.

[20] Sun Micrsystems. *Java Data Objects Specification. Versin 1.0.* Internet: `http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html`, 2003.

[21] Vignaga, A. and Perovich, D. *Enfoque Metodolgico para el Desarrollo Basado en Componentes.* Technical Report RT03-14, InCo Pedeciba, Montevideo, Uruguay, 2003.