APPLICATIONS FOR ECONOMIC ORGANIZATIONS BUILT ON ENTERPRISE JAVABEANS TECHNOLOGIES

Dănuţ-Octavian SIMION, PhD Associate Professor
Athenaeum University, Bucharest, Romania
danut_so@yahoo.com

Emilia VASILE, PhD Professor
Athenaeum University, Bucharest, Romania
rector@univath.ro

Abstract: *The paper presents the role of Applications for economic organizations build on Enterprise JavaBeans technologies. The IT systemS should not only be seen as an interface between the operating system and the management system, but also as the connecting element of the internal environment of the company and its external environment (economic, financial, banking, etc). The main purpose of the information system is to provide every user, according to his responsibilities and responsibilities, all the necessary information. Executive Information Systems (EIS) represents information systems designed to provide: quick and selective access to internal and external data of the company, information on critical success factors determining strategic objectives, calculation facilities and special graphical representations. The business applications rely on databases and on the front side are using different web technologies. To make a connection between these components, the best method is to use Enterprise Java Beans that can map any object inside an application. Enterprise JavaBeans is component architecture. The fields of application and the variety of forms of component architecture can be quite diverse. Enterprise JavaBeans is a rather unique variant: a server-side, transaction-oriented component architecture for distributed components.*
Keywords: *Enterprise JavaBeans, business logic, IT systems, programing logic, informational support, application components, business decisions*
JEL Classification: *C23, C26, C38, C55, C81, C87*

1. Introduction

A business component is a software component small enough to create and maintain in one piece and large enough to provide useful and practical functionality and to justify separate maintenance; it is equipped with standardized interfaces that allow it to cooperate with other components. First, we simplify the view of a component, imagining it as some kind of Lego building block. The interior works of the block remain hidden from view. However, it can be seen that it has connectors that allow it to be attached to other building blocks. A combination of suitable building blocks results in a structure that serves a particular purpose - a house, a garage, a road. Software components are also blocks where one cannot necessarily be seen indoors. Its functionality can only be deduced from the public interface, which in addition to allowing it to be used, also allows connection to other components. As with the Lego building blocks, with the software components the crucial property is the reuse. A component that can be used in a single application scenario is not a genuine component (G.Anderson, and P.Anderson, 2018; Barry and Dick, 2019).

Objects also offer the interface concept, which is usually strictly coupled to the basic system technology and thus limits interoperability. Undoubtedly, the close relationship between objects and components is clear. Thus, object-oriented approach and techniques seem to provide the best basis for component development and component-oriented software. A fundamental concept of the component paradigm is that of the interface. The interface of a component is a kind of contract whose conditions the components are obliged to fulfill. It is a point of interaction with the components, documenting their characteristics and capabilities. A business component can have multiple interfaces. Each interface represents a service provided by the component.

2. The Enterprise JavaBeans component architecture for business components

Enterprise JavaBeans is a component architecture. The fields of application and the variety of forms of a component architecture can be quite diverse. Enterprise JavaBeans is a rather unique variant: a server-side, transaction-oriented component architecture for distributed components. Thus, Beans Enterprise are components that provide services to multiple clients on a single server. Without a framework that incorporates components into a kind of running environment and provides them with the necessary services, each

component that will be made available through a network should have its own server. This would make the development of these components much more difficult and if more components were implemented on a computer it would result in unnecessary strain on its resources. Even the reuse of a component can be endangered, as the servers often have to adapt to the base platform. An architecture of components, such as Enterprise JavaBeans, enables the deployment of components for distributed applications without significantly affecting the components themselves (Jain, 2017; Hanson, 2018).

There is a list of requirements that the architecture of the components must meet:

- Independence of the environment: the components should be implemented without reference to the programming language, operating system, network technology, etc.
- Location transparency: For the component user it should not make any difference if the component offers its services in the user's local address space or in the address space of another computer, remotely. The mechanisms required for the transparent use of local or remote components should be made available through component architecture.
- Interface and implementation separation: The specification of a component must be completely independent of its implementation.

Self-descriptive interfaces: In order to achieve a free coupling of components during running, a component should be able to provide information about its capabilities and entry points.

A JavaBean is essentially a Java class that respects the rules set out in the JavaBeans specification. The most important attributes of a bean are its public interface, the possibility of analyzing it based on its composition, its adaptability to individual requirements and its ability to persist - by serializing objects. The public interface consists of the properties of a bean, the methods it allows others to use, and the events it receives or executes. A bean can be a visible component - a button, for example, or an invisible component - for example, a network service.

Immediate trouble-free usability: A component should be used on any platform without being adapted in any way - which implies a binary independence of the component code.

Integration and composition ability: In combination with other components, a component should be able to contribute to the creation of new usable components (Liu, Bass and Klein, 2017; Monson-Haefel, 2018).

3. Development of business components with Enterprise JavaBeans

However, Enterprise JavaBeans is not just a component architecture. The specification defines a system-oriented component model for the notion of component model. This makes it possible to implement different types of Enterprise bean. Defines protocols for managing the components, for cooperating and communicating the components between them and for their use by a client (G.Anderson and P.Anderson, 2018; Monson-Haefel, 2018).

JavaBean example:

```
public class AValidBean implements AEventListener {
    private int aProperty;
    private Vector beanListeners;

    public AValidBean()
    {
       aProperty = -1;
       beanListeners = new Vector();
    }
    public void setAProperty(int value)
    {
       aProperty = value;
    }
    public int getAProperty()
    {
       return aProperty;
    }
    public void addBEventListener(BEventListener listener)
    {
       beanListeners.addElement(listener);
    }
    public void removeBEventListener(BEventListener listener)
    {
       beanListener.remove(listener);
    }
    private void fireBEvent() {
       BEventListener l;
       for(inti=0;I< beanListener.size(); i++) {
           l = (BEventListener)beanListener.elementAt(i);
```

```
            l.notify(new BEvent(this));
        }
    }
    //Implementation of AEventListener Interface
    public void notify(AEvent event)
    {
        //processing the event
    }
}
```

This bean class is not derived from any class. It does not implement any standard interface and is still a valid JavaBean - only visible JavaBeans must come from java.awt.Component. It simply follows the naming conventions set out in the specification. It has the property of aProperty, which can be manipulated and read by the setAProperty and getAProperty methods. Because it implements the AEventListener interface, it can react to the AEvent event. It triggers the BEvent event, for which other beans can be registered through addBEventListener and can be registered by removeBEventListener. By exchanging events, the bean can dynamically pair with each other, as registration for certain events can be recorded and canceled during running. This coupling over events is also a free coupling, as the bean is extracted from the actual type using the appropriate listener interfaces.

With the naming convention type get property, void property (type), implements EventType Listener, void add EventType Listener () and void eliminates eventTypeListener () etc., a builder can, for example, analyze (introspection) the bean with the help of Java Reflection API in terms of its properties and the possibility to link it to events. The instrument can place the user in a visual bean handling position. Thus, the JavaBeans specification essentially focuses on the description of the program interface for:

- recognition and use of JavaBeans properties,
- adapting JavaBeans to particular circumstances,
- event logging and sending between individual JavaBeans,
- persistence of JavaBeans components.

On the other hand, the Enterprise JavaBean specification focuses on distributed computing and business transactions. JavaBean objects do not have a distributed character. The EJB specification describes a service framework for server-side components. Enterprise Beans are never visible components of the server. You

can look in vain for the EJB specification for a discussion of the properties and events of an Enterprise Bean, as it mainly describes the programming interface and properties of the framework.

Of course, servers can be built on traditional JavaBeans. Then, however, the framework itself should be developed, which provides the components with the relevant server utilities and refers to distribution. However, we could imagine a combination of invisible JavaBeans and Enterprise Beans in which an Enterprise Bean provides a certain interface on the EJB server and delivers calls to JavaBeans - for example, by triggering JavaBean events (Jain, 2017; Barry and Dick, 2019).

We should not try to look for too many similarities between the two models, because, despite a superficial similarity of name, the two models are quite different in emphasis. However, JavaBeans and Enterprise Beans should not be considered as opposing concepts, but rather complementary.

Enterprise JavaBeans (EJB) is a component of the Java Enterprise Edition platform. In this model, EJB takes over the part of the server application logic that is available as components: Enterprise Beans.

These contain the logic of the application used by the client programs. Enterprise Beans are in an EJB container, which makes a running environment available to them so that, for example, they can be addressed by client programs through home and remote interfaces and have the possibility to communicate with each other through the local home and local interfaces, so that life cycle management can be ensured. The EJB container is connected to services through the standard programming interface, services that are available for bean - for example, access to JDBC databases, access to a JTA transaction service and access to a JMS messaging service. The EJB container is installed - possibly next to other containers on an application server.

The Server part

The server is the fundamental component of the EJB architecture. Here we are not deliberately talking about an EJB server. In fact, it should be called J2EE server. Sun Microsystems strategy in relation to enterprise applications within the J2EE platform involves Enterprise JavaBeans to a much greater extent in the complete portfolio of Java-based interfaces and programming products and the Enterprise JavaBeans specification.The Enterprise JavaBeans specification does not define any requirement on the server. The reason for this is probably their stronger integration into the Java Enterprise Edition platform.

A J2EE server is a running environment for various containers (one or more of which can be EJB containers). In turn, each container makes a running environment available for a particular type of component. Java application server creators are increasingly looking to support the J2EE platform. Almost a manufacturer that offers a pure EJB server. In the meantime, many CORBA database providers, transaction monitors, or ORBs have started supporting Enterprise JavaBeans.

In the J2EE platform environment and thus indirectly in the EJB architecture, the server component has the responsibility to provide basic functionalities. This includes, for example:

- Wire and process management (so that several containers can offer their server services in parallel);
- Support for clustering and task sharing (ie the ability to run multiple servers cooperatively and distribute client requests according to the task on each server to get the best response times);
- Security against breakdown (failure-safety);
- A name and directory service (for locating components);

Access and share operating system resources - for example, network sockets for running a web container.

The interface between server and container is highly dependent on the manufacturer. Neither the specification of Enterprise JavaBeans nor that of the Java 2 platform, Enterprise Edition, define the protocol for this. The Enterprise JavaBeans specification in the above version assumes that the server and container manufacturer are one and the same (Liu, Bass and Klein, 2017; Hanson, 2018).

The EJB container

The EJB container is a running environment for Enterprise Bean components. Just as an EJB container is assigned to the server as an execution environment and service provider, a bean depends on its EJB container, which provides it with a running environment and services. Such services are provided to the bean through standard programming interfaces.

A Java application server vendor can provide additional services through the standard interface. Some manufacturers offer, for example, a generic generic service interface for the manufacturer through which specially developed services can be provided - such as a logging service or user management. If an Enterprise Bean uses such services of its own, then it cannot simply be placed in any available container.

The EJB container provides Enterprise Beans with a running environment and also offers particular Enterprise Beans services during running through the static programming interfaces mentioned above. Now we want to examine the most important aspects of both areas - the rolling environment, as well as the services provided.

Enterprise Beans

Enterprise Beans are the server-side components used in Enterprise JavaBeans component architecture. They implement the logic of the application on which the client programs are based. The functionality of the EJB server and container ensures only the use of the beans. Enterprise Beans are installed in an EJB container, which provides them with an environment during which they can run. Enterprise Beans is implicitly or explicitly based on the services offered by the EJB container:

Default in case:

- persistence managed by containers (CMP);
- declarative transactions;
- security.

Explicitly in the case:

- use of explicit transactions;
- persistence of beans (BMP);
- sending asynchronous messages.

Types of Enterprise Beans:
There are three different forms of Enterprise Beans, which differ more or less sharply from one another: entity beans, message-driven beans, and session beans.

Session beans model common processes or events. For example, this could be introducing a new customer into an enterprise resource planning (ERP) system, executing a reservation in a reservation system, or establishing a production plan based on open orders. Session beans can be viewed as an extension of the client arm to the server. This view is supported by the fact that a session bean is a private resource of a particular client.

Entity beans, on the other hand, are real-world objects that are associated with particular data, such as a customer, a booking account, or a product. An instance of a particular type of entity beans can be used simultaneously by several clients. Session beans usually operate on data represented by the entity beans.

Message-driven beans are the recipients of asynchronous messages. A messaging service acts as a mediator between the sender of a message and the bean driven by messages. Entity session beans are addressed via the local or remote interface. Calls to entities or session beans are synchronous; that is, client execution is blocked until the Enterprise Bean method has been processed. After the method call has returned, the client can continue processing. Message-driven bean can be addressed only by the client (indirectly), sending a message on a certain channel of the messaging service. A certain type of message-driven bean receives all the messages that are sent on a specific message service channel. Communication through a messaging service is asynchronous. That is, the execution of the client can continue directly after sending a message. It does not remain locked until the message has been delivered and processed. The container can implement multiple instances of a particular type of message-driven beans. Thus, in this case parallel processing is possible. Message-driven beans have no status between processing multiple messages. In addition, they have no identity with the customer. In a sense, they are similar to session beans without status. To process a message, the message-driven bean can use session or entity beans, as well as all of the services the container offers.

There is another distinction regarding session bean, namely whether or not the session bean is. Stateless session beans do not store data from one method call to another. The methods of a stateless session bean only work with the data transmitted as parameters. Sitting berries without the same type status have all the same identity. Because they have no state, there is neither the necessity nor the possibility of distinguishing one from the other.

On the other hand, the statistical session bean stores data on several methodical calls. Method calls to session bean can change the bean status. The status is lost when the client no longer uses the bean or when the server is disconnected. Session beans of the same type have different identities at run time. The EJB container must be able to distinguish them because they have different states for their customers. A session bean receives its identity from the EJB container. Unlike the entity bean, the identity of a session bean is not visible on the outside. As clients always work with a session bean which is an exclusive instance for them, there is no need for such visibility.

Entity bean can be distinguished by the fact that they themselves are responsible for making their data persistent or if the EJB container takes over this task. In the first case, there is talk of persistence managed by bean, while in the second it is persistence managed by containers. Entity bean of the same

type have different identities at run time. An entity kernel of a particular type is identified during running by the main key, which is allocated by the EJB container. Therefore, it is related to particular data, which it represents in the activation phase. The identity of an entity grain is visible on the outside (Jain, 2017; Monson-Haefel, 2018).

Bean types play a role in managing EJB container resources. With entity bean, message bean and session bean stateless, the recipient can initiate accumulation, while with session bean, it can instigate passivation and activation - serialization and deserialization on a secondary storage medium. The interface between an entity bean and the EJB container is called context (javax.ejb.EJBContext). This interface is again specialized for the three types of bean (at javax.ejb.EntityContext, javax.ejb.MessageDrivenContext and javax.ejb.SessionContext). The bean can communicate with the container using the context that is passed from the EJB container to the bean. The context remains related to a bean for the whole life. In context, the EJB container manages the identity of an Enterprise Bean. With a change in context, the EJB container can change the identity of a bean.

Remote Interface

The remote interface defines those methods that are not offered externally by a bean. The methods of the remote interface thus reflect the functionality expected or required by the components. The remote interface must be derived from javax.ejb.EJBObject, which in turn is derived from java.rmi.Remote. All remote interface methods must declare the exception java.rmi.RemoteException.

```
package ejb.accountbank;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface Bankaccount extends EJBObject
{
   // add account number
   public String getAccNumber ()
     throws RemoteException;
   // account description
   public String getAccDescription ()
```

```
    throws RemoteException;
  // balancing cont
  public float getBalance ()
    throws RemoteException;
  // increase balance account
  public void increaseBalance (float amount)
    throws RemoteException;
  // balance reduction account
  public void decreaseBalance (float amount)
    throws RemoteException;
}
```

Home Interface

The home interface must be derived from javax.ejb.EJBHome - in this interface is the method of removing a bean; it should not be declared separately. EJBHome, in turn, is also derived from javax.rmi.Remote.In the home interface, also all methods declare the java.rmi.RemoteExeption exception to be triggered. As in the case of the remote interface, everything indicates the distributed character and the incorporation within the EJB.

```
package ejb.accountbank;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface BankAccountHome extends EJBHome
{

  // generate account
  public BankAccount create (String accNo,
                             String accDescription,
                             float initialBalance)
    throws CreateException, RemoteException;


  // find a specific account
  public BankAccount findByPrimaryKey (String accPK)
    throws FinderException, RemoteException;
}
```

Bean Classes

Bean classes implement the methods that have been declared in the home and remote interfaces (except for the findByPrimaryKey method), without actually implementing these two interfaces. The signatures of the remote and home interface methods must be in accordance with the appropriate methods of the bean class. The bean class must implement an interface that depends on its type, and it must be javax.ejb.EntityBean, javax.ejb.MessageDrivenBean or javax.ejb.SessionBean. The bean does not implement either its home or its remote interface. The summary of the class is only for an entity bean with automatic persistence managed by containers. Session, message and entity bean classes, which manage their own persistence, are concrete classes.

```
package ejb.accountbank;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.RemoveException;

public abstract class BankAccountBean implements EntityBean {
  private EntityContext theContext;

  public BankAccountBean () {
  }

  // method of creating the home interface

  public String ejbCreate (String contNr,
                                  String accountDescription,
                                  float initialBalance)
    throws CreateException
  {
    setContNumar (contNr);
    setContDescriere (contDescriere);
    setContBalance (initialBalance);
    return null;
  }

  public void ejbPostCreate (String contNr,
```

```
                                    String accountDescription,
                                    float initialBalance)
  throws CreateException
{
}

// abstract getter / setter method

public abstract String getContNumar ();
public abstract void setContNumar (String cnr);

public abstract String getContDescriere ();
public abstract void setContDescriere (String cnr);
public abstract float getContBalance ();
public abstract void setContBalance (float acb);
// the remote interface methods

public String getCntNumber () {
  return getContNumber ();
}

public String getCntDescription () {
  return getContDescriere ();
}
public float getBalance () {
  return getContBalance ();
}
public void increaseBalance (float sum) {
  float acb = getContBalance ();
  acb + = sum;
  setContBalance (CBA);
}
public void decreaseBalance (float sum) {
  float acb = getContBalance ();
  acb - = sum;
  setContBalance (CBA);
}

// the methods javax.ejb.EntityBean interface
```

```
  public void setEntityContext (EntityContext ctx) {
    theContext = ctx;
  }

  public void unsetEntityContext () {
    theContext = null;
  }

  public void ejbRemove ()
    throws RemoveException
  {
  }

  public void ejbActivate () {
  }

  public void ejbPassivate () {
  }

  public void ejbLoad () {
  }

  public void ejbStore () {
  }
}
```

For Java programming language and its components, talk is always about beans. The most popular current component of Java is most likely JavaBeans. Before we get into the difference between JavaBeans and Enterprise JavaBeans, we need to briefly discuss the component world, without wishing to engage in a full discussion of the component paradigm.

4. Conclusions

Descending strategy, top-down, is based on the principle of decomposing the complex computer system into components, presenting a lower complexity (defined by fields of activity, for example), successively through several levels of detail within each defined component. Through this approach, the computer system acquires a hierarchically modular structure in which each component

fulfills a certain functionality and will be coordinated in its operation by the components placed at the immediately higher hierarchical level. (G.Anderson, P.Anderson, 2018; Monson-Haefel, 2018). Upward strategy, bottom up, promotes the initiative at the level of each management area (accounting, business, production, etc.) without a defined framework and architecture for the global IT system at the organization level. The management systems are designed, realized and exploited independently, responding to the management requirements of the domains for which they were created, and will subsequently be integrated into the overall IT system of the organization (Jain, 2017; Monson-Haefel, 2018). The Enterprise JavaBeans are the optimal solutions when a business application needs to access the main components that define the economic logic. The entities that are residing in the database store data that is needed in the business flows and the easy way is to use Enterprise JavaBeans for the enterprise applications.

References

Anderson, G. and Anderson, P. (2018). *Enterprise JavaBeans Component Architecture: Designing and Coding Enterprise Applications*, Prentice Hall.

Barry, D,K., and Dick, D. (2019). *Web Services, Service-Oriented Architectures, and Cloud Computing: The Savvy Manager's Guide*, second edition. Barry & Associates, Inc.

Hanson, J. (2018). *Enterprise JavaBeans: A Primer*. SitePoint Publishing.

Jain, A. (2017). *How to work with Enterprise JavaBeans (EJB)*. MrBool Publishing.

Liu, A., Bass, L., and Klein, M. (2017). *Analyzing Enterprise JavaBeans Systems Using Quality Attribute Design Primitives*. Carnegie Mellon University.

Monson-Haefel, R. (2018). *Enterprise JavaBeans.* O'Reilly & Associates.