Journal of Computer Science 2 (11): 807-814, 2006
ISSN 1549-3636
© 2006 Science Publications

The Design of a Complex Software System with Archjava

[1]Djamal Bennouar, [2]Tahar Khammaci and [3]Abderrezak Henni
[1]Saad Dahlab University, Route de Soumaa, 09000 Blida, Algeria
[2]LINA, Université de Nantes, 2, Rue de la Houssinière, 44322, Nantes, France
[3]Institut National d'Informatique, Oued Smar, 16000, Algiers, Algeria

Abstract: Software development by assembling components represents a very promising way for the design of high quality software at lower costs. The assembly is specified by the Architecture Description Languages (ADL). ArchJava is an ADL that is characterized by its independence from the application domain and its close position to the implementation level, thus making it very attractive for practitioners. Until now, ArchJava was used to illustrate its characteristics either on simple cases, or for the software specification at a high level of abstraction. In this study we present the design of a complex software system by assembling components using ArchJava and following a top down design process. This experience underlines the power of design by assembling components for the fast realization of complex software system. It also emphasizes some deficiencies of ArchJava in the specification of some concepts related to the design by assembling components, the design of the GUI and data modeling.

Key words: Software architecture, connector, component, ArchJava

INTRODUCTION

Over the years, the object model has been successful in various fields of software system design (RAD, Distributed Object Infrastructure, Component models such as EJB, CCM, .Net etc.). However, in the last decade the software engineering community has made significant progress in reducing the semantic gap between the mental model of the software architect and the models handled by software tools (diagram, program etc). This progress was materialized by the emergence of software architecture as an autonomous field of research/development in software engineering. The purpose of software architecture is to provide the concepts, techniques and tools needed to deal directly with various aspects of mental models related to a software system.

Historically, a mental model has always been expressed in an informal box and line diagram notation. Each box deals with a precise functionality of a system. The lines correspond to interactions semantics or data flow. This view of software is often called software architecture and often represents the first step leading to the realization of a software product. Furthermore, software architecture aims to easily and efficiently accommodate the mental models of architects and tries to join other engineering areas, by adopting the strategy of designing a system by assembling different components. To reach this objective, the procedural paradigm of interaction which prevails in the object model and the component models like JavaBeans, ActiveX and EJB must be completely abandoned (as was the case with structured programming with the emergence of the object model).

A number of research initiatives were undertaken in the last decade concerning software architecture specification. These efforts resulted in the proposal of a great number of ADL. The work presented in[1] summarizes the characteristics of these ADL and discusses the main concepts introduced in Software Architecture such as components, ports, composite components or configurations and connectors. Recently, UML 2.0[2], in an attempt to fill the gaps of UML 1.4, has introduced some mechanisms in order to support the concepts just introduced. Still, ADL has not been highly successful. This is due to several factors, such as the orientations to solve problems in a specific domain[3], the use of a particular architecture style[4], or the exclusive use of formal languages like CSP[5].

ArchJava[6] was introduced to overcome the stated deficiencies. It offers an attractive alternative for the practitioners: an ADL, at the implementation level, supporting the main concepts needed for the design by assembling software components.

ArchJava is an extension of the JAVA language. It brings to java the fundamental concepts of software architecture, namely the concepts of components, ports, connectors and composite components. ArchJava was tested on simple cases or to make specification at a high level of abstraction[7], with an aim of illustrating the various concepts which it handles. In the present study, we use ArchJava to realize a complex software system intended for a telecommunication company. The software system manages the commerce of various products of an X25 network. Moreover, this study

Corresponding Author : Saad Dahlab University, Route de Soumaa, 09000 Blida, Algeria

determines the strong and weak points of ArchJava when it is used to design concrete software system.

THE ARCHJAVA LANGUAGE

ArchJava was designed to allow system design by assembling components. Significant work was done to ensure the validity and the integrity of the interactions between components in ArchJava. These interactions are done only through connectors. ArchJava supports the following concepts of software architecture:

Components and ports: An ArchJava component is a special object, being able to be equipped with ports through which it can interact with other components. It is declared by the *component class* keyword. A port corresponds to the concept of interface found in COM or CCM models. It is declared by the keyword *port*. At the port level, a component exposes a set of methods organized in two main categories:

* **Provided methods:** These methods are implemented by the component and can be activated by other components. A provided method is declared by the keyword *provide*.
* **Required methods:** These methods are implemented by other components and are declared by the keyword *require*. A component reaches an external service by activating a required method of a port.

In spite of the possibility offered by a port to mix the provided and required methods, in practice it is recommended to put in a port, either the required methods or the provided methods. The basic rules of the establishment of connection between methods of ports are as follows:

* A provided method is connectable to a required method.
* A provided method may be connected to several required methods.

Composite component: A composite is composed from many interconnected other components which can either be composite or non-composite. The ports of the composite are connected to the ports of its internal components by special connectors named *delegation connectors* in UML 2.0 or *glue* in ArchJava. With the concept of composite component, software architecture has a hierarchical structure. Its development follows a top down process which can comprise several stages of refinement. In the next sections, while needed, we will schematize *component*, *ports* and *connectors* according to the recommendations of UML 2.0.

TARGETED APPLICATION AND ITS DOMAIN

In this experimental study, we plan to realize a software system to manage the commerce of products of an X25 network for a telecommunication company.

A telecommunication company may have one or more inter-connected X25 sub-networks. Each sub-network may be provided by an independent manufacturer. Each manufacturer equips the sub-network that it provides with a control station. The control station, in addition to the various actions of control of the sub-network, collects tickets describing communications over the network from the sub-network switches. A second control station is also provided for each X25 sub-network. This second station is used in case the first control station fails to work. The second station carries out the collection of the tickets in parallel with the first station. The tickets collected by the second station are exploited only in the event of breakdown of the first station

A subscription with an X25 network is determined by a number similar to the telephone number. A customer can have one or more subscriptions. The tickets inform about the customers and the products which they consumed during one period of a precise connection. As an example, we can find on a ticket the caller number, the called number, the date and time of starting a connection, the duration of a connection, the quantity (in bytes) of transferred data, etc.

The major goals of the application are the management of X25 network products, subscribers, subscriptions, invoicing, payment, tariff plan, system maintenance, and users. The application downloads a file containing tickets from a selected control station of the X25 sub-network. The tickets are evaluated according to a tariff plan and the information provided by each ticket. In addition to the information provided by the tickets, other information will be used for the calculation of the invoice of a subscription for a period. Among this information we find the speed of transfer of data, the hiring of modems, the membership to a secure and closed group of subscribers, etc. Other details concerning the application will be given in the next section.

THE DESIGN PROCESS

The design process we follow is characterized by the following elements:

* It is a strongly oriented design by assembling components. It consists of developing a tree representing the hierarchy of refinement of the system. The root of the tree corresponds to a component representing the system to be realized. The leaves represent the not composite components which contain only the ports declaration and the realization in java of provided and internal methods. The intermediate nodes represent composite components. The design process follows a top down strategy of development. Thus the design will begin with the global definition of the component at the root of the refinement tree.

* It is iterative. At each design stage, several attempts of modeling are evaluated. The decisions at a design stage can lead us to reconsider the results of the preceding stages.
* It is recursive. The process at the current stage is applied to the following stage.

A composite component may contain a special component known as controller. Among the roles assigned to the controller are the initialization of the composite, the management of the flow of control and the multiplexing of ports. The process comprises a number of stages. Each stage corresponds to a level of refinement represented by the refinement tree. Each stage has two phases. In the next section we present a partial view of this process. Complete description can be found in[8].

Phase 1: The External view

In this phase the following actions are performed:

Global description: This phase begins with an abstract definition of the system or component. In a pictorial sense, it is in the form of only one box that has several arriving and leaving arrows. Arrows represent the provided and needed services, data or control. To eliminate any ambiguities, this box and arrow specification must be accompanied by a narration explaining the total functionalities of the system and the semantics associated with the arrows. The narration may also contain the requirements and constraints fixed by the customer.

Organize the services: From the preceding definition, we must define:

* The provided services
* The required services

Definition of adaptation services: These are internal services. The main role of an adaptation service is to manage connections to legacy systems or system not realized in ArchJava.

Definition of the external ports: The required and provided services are gathered in ports. A port will contain either the provided services or the required services.

At the end of this first phase, a global ArchJava description of the system becomes possible.

Phase 2: The Internal view

In this phase, the following goals must be achieved.

Define the internal system logic in a box and line style.

Define the roles played by the controller. Among these roles we can find the composite component initialization, the logic of activating components according to a defined strategy (sequence activation, parallel activation etc...), the multiplexing of method between external and internal ports and the management of the composite component environment

Define a mapping between external services and the internal box: This mapping will show exactly which component will realize a provided method and which one need a required method exposed on external ports. This mapping will serve to determine if a port multiplexing is needed and to realize the delegation connectors linking external ports to internal components ports.

In order to achieve this mapping, we have to apply one of the following strategies.

* All methods, in an external port, are mapped to only one component. In this case, we will find a similar port in the internal component.
* Methods in an external port are mapped to more than one component. To realize the connection of external method to a method in a port of an internal component, we have to use the multiplexing technique which will be held by the controller component. The external port is then connected to a similar port of the controller.

Internal component design: For each box of the internal view, one of the following actions has to be performed

* Find an existing component which can realize all of the objectives assigned to a box and use it. In case where no existing component corresponds to the box objectives, apply the two design phases to the box in order to completely define the external view (ports) and internal view (component and connections)
* Realize the delegation connectors
* Realize the connection using existing connector technology. The process we present here does not deal with the engineering of connector's architecture.

DESIGN OF THE TARGETED APPLICATION

In the following, we will apply the just described process to the design of the targeted application previously presented. We present some aspects of this process in order to illustrate how software architecture is conducted with ArchJava

Phase 1: The external view

Global description of the system: Figure 1 shows the global view of the system. It is clear that such a view is not very precise and is ambiguous. It requires a complementary narration and could not be treated by software tools

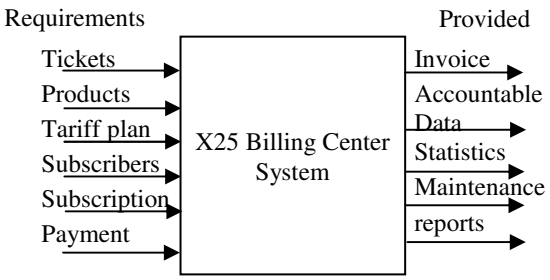

Fig. 1: The global view of the system

System ports: The preceding view is transformed into a more precise view which would be the starting point for a successive operation of refinement until reaching the desired software product. Figure 2 shows all the ports of the system. A number of methods dealing with some objectives assigned to the port are defined within each port. Note that all the functionalities quoted in the previous informal definition don't appear in Fig. 2. It is the case of the ticket acquisition service. This service will be defined in the internal view of the system because it connects our system to another system, not defined in ArchJava

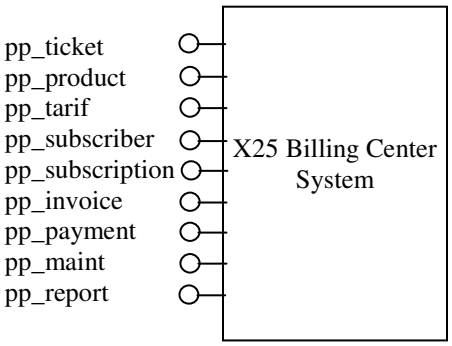

Fig. 2: System ports

Adaptation services: These are the services (presented in the previous informal view) that are needed by the system we are designing, but are provided by other systems which we will call foreign systems or foreign component. The external view of a foreign component or system is not compatible with the external view of the system or component designed in ArchJava. In such cases, the following rule must be applied: When a component or a system defined in ArchJava interacts with a foreign system, connection to the foreign system has to be established through an adapter component (Fig. 3). This one interposes between our system and the foreign system. The adapter will be given the responsibility to hide the specifics of the foreign system.

ArchJava description: Figure 4 shows, in ArchJava, the external view of the system we plan to realize. Henceforth, this system will be named X25BillingCenter.

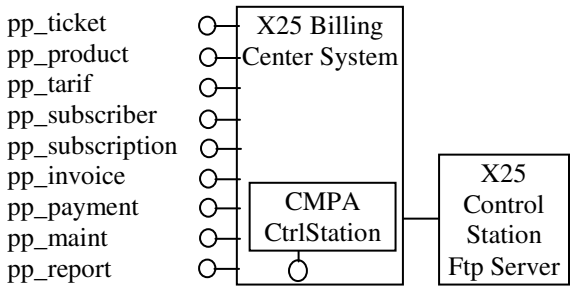

Fig. 3: Connection to a foreign system

```
public component class  X25BillingSystem
 {
// 1- External View Start Here----------
// 1- Provided services
// Not all port will be shown here
  public port  pp_subscriber{
       provide  int   subscriberAdd (
                 Subscriber   subscriber);
       provide  int subscriberUpdate (
                    Subscriber subscriber);
       provide  int subscriberDelete(
                  Subscriber subscriber);
    }
   public port  pp_subscription
            provide  int  subscriptionDefine (
            Subscriber   subscriber,
            Subscription subscription);
            provide  int  subscriptionRelease (
            Subscription subscription);
            provide  int  subscriptionUpdate (
            Subscription subscription);
            provide  int  subscriptionStart (
            Subscription subscription);
            provide  int  subscriptionStop (
            Subscription subscription);
    }
    // Other provided ports go here
   // 1-2 –  Required Port
        // No Required port for this system
   // External  View  End  Here ---------

   // 2- internal View
   // 3- Activate the controller component
public void run() {
   controler.pp_starter.activate()
}
   // 4-  Next method ensures that  the  component may  be
  // operated in a stand alone mode
public static void main(String[] arg){
                    new  X25BillingCenter().run();
}
```

Fig. 4: External view of the system

Phase 2: The internal view

System logic in a box and line style: Figure 5 shows the fundamental components of the X25BillingCenter's internal architecture. We describe in what follows some components. A complete description is given in[9].

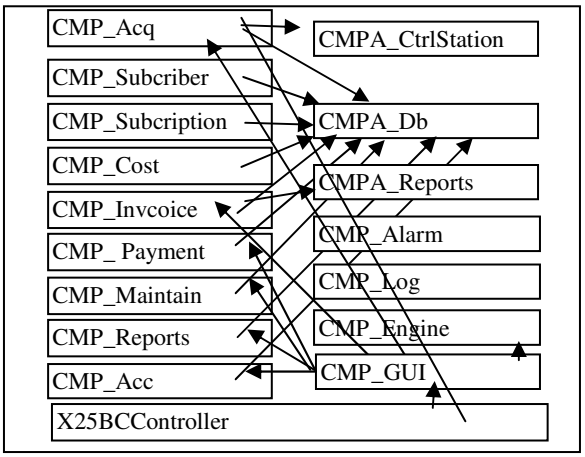

Fig. 5 Box and line style

CMP_acq will assume the responsibility of bringing back the tickets from the control station, format them, valorize them and finally send them to a data base. The tickets in the database will be used thereafter for the calculation of the invoices. This component will require at least the services of two adapters of the foreign system: The first adapter will interpose between the control station and the system to be realized. The second will interpose between a database and the system.

CMP_Subscriber provides facilities to put various kinds of data concerning the subscribers in databases. This component will use the database adapter component and will be used by the graphical user interface component (CMP_GUI).

CMP_Subscription. A subscription is an X25 number, or address associated with various services and time of validity. A subscription is affected to one and to only one subscriber. A customer may have several subscriptions. This component has the responsibility to define the subscriptions, to assign them to subscribers, to modify, cancel, stop, start and restart them etc. This component uses the services of the database adapter (CMPA_db)

CMP_Invoice carry out the calculation of subscriber's invoices. It requires the services of the data base adapter component (CMPA_Db) and a report tool component adapter (CMPA_Reports)

CMP_GUI Allows the interactive and graphical use of the system. It requires the services of components such as CMP_Acq, CMP_Subscriber and so on.

CMP_Engine: Some operations require a time for their termination, which can be very significant. This is the case for the ticket acquisition operation done by CMP_acq component and the invoicing of a significant number of subscribers done by CMP_Invoice. Components like CMP_Acq and CMP_Invoice must have the capabilities to operate independently from other components, in particular the CMP_GUI. The component CMP_Engine was introduced to control, in an autonomous way, the management (starting, stopping, logging etc.) of time consuming operations. It provides the service of configuration (activate/deactivate engine) and requires the services of heavy driven components such CMP_invoice or CMP_Acq

ArchJava description of composition: With ArchJava, it is possible at this early stage to describe the composition of the system by only quoting the different components used (Fig. 6).

```
public component class X25BillingSystem
 {
   // 1- Internal View: Composition
   private final CMP_Acq acquire =
                    new CMP_Acq ( );
   private final CMP_Subcriber subscriber =
                    new CMP_Subscriber ( );
   private final CMP_Subcription subscription =
                    new CMP_Subscription ( );
//..
// … other component used must be specified here
// ..
       private final CMP_Alarm alarm =
                    new CMPAlarm ( );
     // Next component is the system controller
      private final X25BCController controller =
                    new X25BCController()
    // End of Internal View System Composition
     // 2- External View
     // 2- 1 Provided services
```

Fig. 6: Specification of needed component for the internal view

Mapping external ports to components: In this first stage of refinement, it appears that all methods of each port will be mapped to one component since all ports contain only the provided services. All methods of each port will be implemented by only one component. Thus the mapping is direct and no multiplexing functionality is needed. As a result, in each component that is mapped to an external port we will find a similar port. As an example, all the methods of the external port pp_ticket will be implemented by CMP_Acq. Thus, CMP_Acq has to provide a similar port which we name pp_ticket. Figure 7 shows a partial mapping, materialized by delegation connectors.

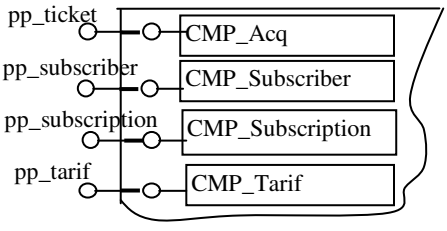

Fig. 7: Mapping external ports to internal ports

The design of internal components: For each component of this stage of design, it is necessary to search if a component which carries out the same objectives exists. In this case it should be used. If this research is unfruitful for various reasons, the design of the component should then be carried out. This lead us to apply to the component the two phases of the design process. In our case, we face a situation where no ArchJava component exists for the telecommunication domain. We are thus brought to realize all our components. In the following, we illustrate the application of the process with its two phases on the CMP_Acq heavy component. .

CMP_Acq external ports: Figure 8 shows all the external ports of component CMP_Acq and Fig. 9 shows the external view described in ArchJava

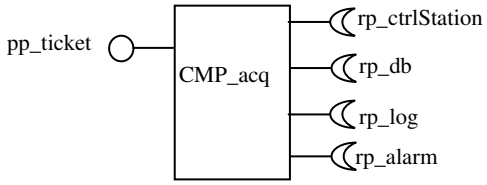

Fig. 8: CMP_Acq external ports

```
public component class CMP_acq
 {
   // 1- Internal View Composition
   // 2- External  View
    // 2- 1- Provided services
    public port  pp_ticket{
              provide  int acquire();
              provide    int format();
              provide    int valorize();
              provide    int clean();
     }
    // 2-2 – Required Services
    public port  rp_ctrlStation
              require  X25Network select();
              require  Station select(
                          X25Network  x25net)
              require    String download(
                         Station station);
     }
    public port  rp_db
              require  int dbconnect (
                    Dbserver  dbserver);
              require   int  dbdisconnect(
                     Dbserver dbserver);
              require  QueryRes Query(
                        String query)
              require   int download(Station station);
     }
 // 3- internal View  Connexion and Glue
 // 4- Below code for Controller Activation
 // 5-  Below code for stand Alone comp
```

Fig. 9: **CMP_Acq** external view in ArchJava

CMP_Acq Internal view: Figure 10 shows a draft of the internal view of this component. It is made up of 4 components and a controller. We present in the following the **CMP_Acq** internal components.

CMP_TicketIn downloads the tickets using the services provided by the adapter CMPA_CtrlStation through the delegation connector connected to the external port rp_ctrlStation.

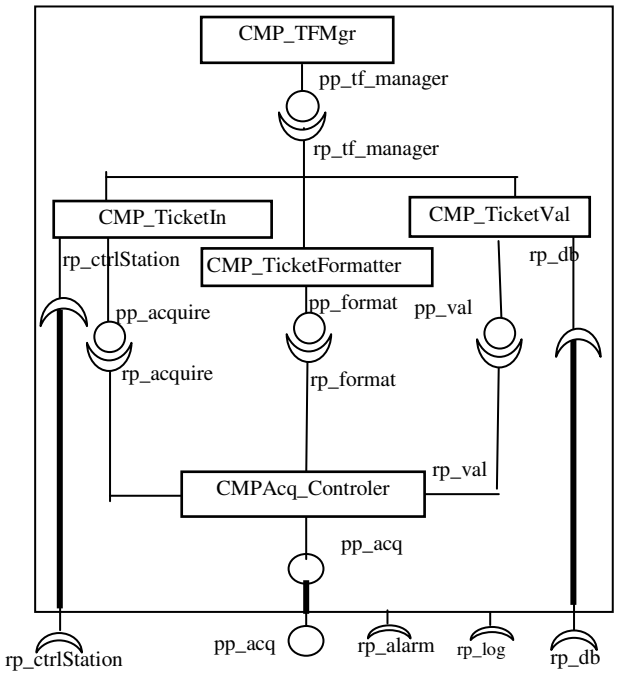

Fig. 10: Component CMP_acq internal view (Not all connexions are shown)

CMP_TicketFormatter transforms a raw ticket in an internal format. This component ensures the independence of the system with respect to the format of tickets specific to each X25 networks manufacturer

CMP_TicketVal. The tickets in the internal format are submitted to this component in order to be valorized. Valorization process is based on the information contained in the ticket (connection duration, quantity of data transferred), on the tariff policy and the services used during the communication described by the ticket. The number of fields representing the ticket are increased in the valorization process by additional fields containing several ticket cost according to tariff policy. After valorization, the tickets are sent the database through the rp_db port.

CMP_TFMg manages the operation on the files containing the ticket in manufacturer format or internal format. It contributes to reinforce the recovery of the operations of various components in case of failure or major error. The provided services are used by the previous components.

CMPAcq_controller: It ensures the activation of the three components CMP_TicketIn, CMP_ TicketFormatter and CMP_TicketVal, as well as the recovery in the event of major problem having stopped the work of one of the three components. It may launch the three components in sequence or parallel and it

carries out the multiplexing of the port pp_ticket. Figure 11 shows the composition description of the CMP_Acq component.

```
public component class  CMP_acq
 {

    // 1- Internal View Comoosition
     private final CMP_TicketIn  ticketin =
            new CMP_TicketIn ( );
     private final CMP_TicketFormater formatter =
             new CMP_TicketFormatter ( );
     private final CMP_TicketVal  ticketVal=
            new CMP_TicketVal ( );
     private final CMP_TicketFileMgr     tfmgr =
            new CMP_TicketFileMgr ( );
     private final CMP_AcqControler  controler =
            new CMP_AcqControler ( );

   // 2- External  View
   // 3- internal View  Connexion and Glue
   // 4-  Below code for Controller Activation
   // 5-  Below code for stand Alone component
```

Fig. 11: CMP_Acq composition

Mapping external ports of CMP_Acq to its component: The mapping between external ports and the internal component port are realized in ArchJava by the glue instruction (Fig. 12)

```
public component class  CMP_acq
 {
  // 3-1 Glue Implementation
  glue  pp_acq  to  controller.pp_acq;
  glue  rp_ctrlStation  to  ticketIn.rp_ctrlStation;
  glue  rp_db  to  ticketVal.rp_db;
  glue  rp_alarm  to  ticketVal.rp_alarm,
                      ticketIn.rp_alarm,
                      formatter.rp_alarm,
                      ticketFileMgr.rp_alarm,
                      controler.rp_alarm ;
  glue  rp_log  to     ticketValuer.rp_log,
                       ticketIn.rp_log,
                       formatter.rp_log,
                       ticketFileMgr.rp_log,
                       controler.rp_log ;
   glue  rp_db  to  ticketVal.rp_db;

  // 3-2 Componenet connection
  // 4-  Below code for Controller Activation
  // 5-  Below code for stand Alone component
```

Fig. 12 : CMP_acq, external to internal ports mapping

CMP_Acq Component connexion: The connection description in ArchJava are specified by the connect instruction (Fig. 13) which must obey to a previously specified connect pattern definition

LESSONS AND COMMENTS

The defined design process was applied successfully for the realization of the X25BillingCenter software. Work carried out showed the flexibility and the power of ArchJava to easily describe architectures specified, even in an informal, ambiguous and incomplete manner. Compared to another work whose goal was to realize the same product using EJB technology[10] the realization time in ArchJava was the shortest. The use of EJB requires the control of several technologies (Servlets, JSPs, JavaBeans, EJBs, XML etc.) and concepts, whereas in ArchJava, the only concepts to be controlled are the fundamental concepts of the software architecture expressed in the JAVA language. A factor which could explain this performance would be the efficient support of the composition concept in ArchJava. This concept made it possible to follow a top down design process. The composition concept is not efficiently supported by the EJB component model[11].

```
public component class  CMP_acq
 {  // 3-2 Componenet connection
  connect  controller.rp_acq, ticketin.pp_acq;
  connect  controller.rp_format,
             formatter..pp_format;
  connect  controller.rp_val, ticketValuer.pp_val
  connect  tfiMgr.pp_tf_manager,
             ticketin.rp_tf_manager,
             formater.rp_tf_manager,
             tiocketVal.rp__tf_manager;
  // Remaining connect expression go here
  // 4- Below code for Controller Activation
  // 5-  Below code for stand Alone component
```

Fig. 13: CMP_acq connectors description

ArchJava represents an important progress in ADL technology. It constitutes an excellent first bridge to practitioner to pass from the object oriented design world towards the design by assembling components world. However, ArchJava encounters some difficulties. We will quote in the following those met in this work.

Connector deficiencies:

* The connection points in a port are represented by methods. It is not possible for the moment to specify another form of connection point (i.e. connection point supporting synchronous or asynchronous mode using shared variable). By limiting the connection point to methods, the ArchJava language did not arrive to release itself from interaction based on procedural model while designing the software system. To reach a degree of maturity and efficiency similar to the other fields of engineering, the design by assembling software component must abandon the paradigm of objects and its communication based on method invocation.
* It is not possible to connect ports having a different number of interaction points (methods), or to connect directly interaction points. This deficiency constrained us in introducing the multiplexing component.
* The connection points of two connected ports must have the same name. This deficiency goes against the significant objective of software architecture, namely, the design of component independently from the environment of use.

* It is not possible to fix default values (default method in our case) on the connection points. The default values (here a default method) are used when the connection point is not explicitly connected. .

Design deficiency: For the CMP_GUI component, the refinement process was very delicate. Thus, in this first version, component CMP_GUI is not composite. It was completely designed with java/swing. The integration of swing component with ArchJava requires the support for asynchronous event connection points and needs the use of the connect instruction to establish connection instead of the event connector model of the swing components. These improvements are possible but require a significant amount of work. This work is concentrated on two essential points:

* Realization of the components adapter for swing components.
* Realization of a personalized connector managing the asynchronous events. This may be achieved by the use of a reflection mechanism provided by ArchJava.

Data modeling: Like any ADL, ArchJava is oriented towards the specification of components and their interactions. ArchJava does not offer any mechanism to model or organize data (i.e. no mechanism to achieve component persistence). Data describing an information system or any real world are not considered as architectural elements. Rather, they feed software architecture and flow in it. Their modeling is carried out in a way independent of architecture. The link will be made through an adapter which is aware of the adopted data model.

CONCLUSION

The work described in this study, presents an actual experience where an approach of designing a complex system by assembling components is used. This experience showed how a software architecture approach could lead to the fast realization of a complex software system. It also raised certain weaknesses of ArchJava and ADL in general. A graphic tool for software architecture specification[12] can fill a good part of these deficiencies by the use of adapters.

This work also raised the need to model independently the software architecture and the data flowing in this architecture. It is obvious that the data could have an impact on architecture, in particular the paths used (connectors). The opposite is also true. Reducing the impact of data on architecture and the opposite would work to enhance the reusability of component in any environment.

REFERENCES

1. Medvidovic, N. and R.N. Taylor, 2000. A classification and comparison framework for software architecture description languages. IEEE Trans. Software Engg., 26: 70-93.
2. OMG, Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04, http://www.omg.com
3. Vestal, S., 1993. Scheduling and Communicating in MetaH. Real-Time Systems Symp., pp: 194-200, Raleigh-Durham (NC).
4. Medvidovic, N., R.N. Taylor and E.J. Whitehead, 1996. Formal modeling of software architectures at multiple levels of abstraction. Proc. California Software Symp., pp: 28-40, Los Angeles, CA.
5. Allen, R.J., 1997. A formal approach to software architecture. Ph.D. Thesis. School of Computer Science, Carnegie Mellon University.
6. Aldrich, J., 2003. Using Types to enforce architectural structure. Computer Science and Engineering, Ph. D. Thesis, Washington University, http://www-2.cs.cmu.edu/~aldrich/papers.
7. Aldrich, J., C. Chambers and D. Notkin, 2002. Architectural reasoning in ArchJava. Eur. Conf. Object Oriented Programming, Málaga, Spain, Jun. 10-14.
8. Bennouar, D., 2005. A semi-automated design process for software architecture. RRI04/008, LDRSI Lab, CS Dept., The Saad Dahlab University, Algeria, (In French).
9. Saadi, A., 2005. ArchJava components data book for telecommunication domain. M. Sc. Thesis, LDRSI Lab, CS Dpt.,The Saad Dahlab University at Blida, Algeria (In French).
10. Zougagh, M., 2005. Applying CATALYSIS process to the design of a complex system in telecommunication domain. M. Sc. Thesis, LDRSI Lab, The Saad Dahlab University at Blida, Algeria (In French).
11. Carrez, C., 2003, Behavioral contract for components, Phd Thesis, INFRES, Télécom Paris ENST, (In french).
12. Khider, H., 2005. An IDE for rapid software architecture design with archJava. M. Sc. Thesis, LDRSI Lab, CS Dept., The Saad Dahlab University at Blida, Algeria (In French).