Experiences of Creating COTS Components when Automating Medicinal Product Evaluations

Radmila Juric, Stephen Williams

Cavendish School of Computer Science, University of Westminster,
115 New Cavendish Street, London W1W 6UW, United Kingdom
juricr@wmin.ac.uk, williast@wmin.ac.uk

Abstract. The paper reports on experiences of designing and implementing component based software architecture for automation of medicinal product evaluations across different countries and their evaluation practices. Our generic architectural model renders a set of families of software components that implement required functionality and are potential candidates for COTS components. We identify which role such COTS components may play when populating our software architecture and outline our approach of generating them. We discuss their main characteristics and advocate that such COTS components are developed with a specific component platform in mind and adhere to constraints of our software architecture.

1 Introduction

There is a great interest in reusable software components, particularly commercial-of-the-shelf (COTS) components…… Capitalising on third party expertise and synthesising components technologies with COTS components might bring new answers to problems of interoperability when building today's software systems.

Section 2 details related background and outlines our previous work on automation the medicinal product evaluations. In section 3 we raise the issue of deploying COTS components with component technologies within our problem domain in order to address the interoperation of evaluation practices. We summarise the aim of this paper and comment on any related work in the field. Section 4 describes our process of creating, i.e. extracting COTS components when modelling the running example as an EJB application. The creation of our own design patterns and design decisions dictated by the choice of a component technology has created COTS components that address the interoperability between evaluation practices across various regulatory authorities. Section 5 lists characteristics of such COTS components. We conclude in section 5 briefly discuss our future works.

2 Related Background

Medicinal product evaluation is one of the most important tasks undertaken by government health departments and their regulatory authorities in every country in the world. The independent evaluation of medicinal products is centered on regulations and guidelines for reporting and evaluating data on medicinal products' safety, quality and efficacy. These procedures are strictly defined in order to ensure that all standards on testing, manufacturing and controlling medicinal products are achieved. However, this complex task faces two major problems today. The first one is related to the process of harmonization of different regulatory systems throughout the world as defined in the International Conference of Harmonization (ICH) [1]. Each country has its own system of evaluating medicinal products, which differ amongst themselves not only in vocabulary and definitions of medicinal products, but also in different organizational structures and evaluation practices of individual regulatory authorities. This represents a serious drawback for efficient local and worldwide registration of medicinal products. The second problem is related to automation of such evaluation procedures, i.e. software support in evaluation of medicinal products is a critical task that can dramatically improve the efficiency of regulatory authorities. Furthermore, such software solutions that offer the interoperation of regulatory systems across the world will immediately enhance the ICH harmonization efforts.

In our previous works [1,2] we have analyzed this complex task through local needs of **regulatory authorities** and have extracted their common practices that exist across the world, which is essential if any interoperation between regulatory systems and their evaluation practices will take place. Any software solution that automates such evaluation practices is a large-scale distributed application that requires sharing of data stored in various databases and sharing of processes associated with evaluations [3]. Our starting point was to create a software architectural model, which is layered and component based, and which allows (a) a dynamic generation of applications for medicinal products evaluations and (b) carrying out any evaluation procedure on the application. Such procedures must suit any **regulatory authority**, i.e. regulatory authorities may apply their own evaluation procedure or any other available internationally [3, MI].

In Figure 1 we show the generic architectural model, which is layered and component based. Each layer has different responsibilities. The *application la*yer provides a basic GUI functionality and controls interaction between users and any other layers within the system. This includes the best pathway, i.e. the right choice of R_i and E_i components involved in a particular application. The *domain layer* consists of two families of components. The R_i family of components contains a set of *rules* that are to be followed if we want to have an automated application submission within a particular regulatory authority. The R_i family may also include any future set of rules originated within the ICH. The E_i family of components contains all available *evaluation procedures* that originate in different regulatory authorities or can be found within future harmonised activities from the ICH. Components from the *domain layer* use various data repositories $DB_{1,2,3,4,i,j,k,n}$ stored within components of the *persistence layer*.

Our *persistence and domain layers* can be seen as a common repository of data and processes, where various applicants, regulatory authorities, hospitals, GPs, patients etc., can share data and services defined in our component architecture.

To illustrate the proposed architecture we use a running example, whose functionality is divided into two workflows (see (a) and (b) in section2): (i) submitting an application for evaluating a medicinal product under local regulatory authority rules (R_i) (ii) applying an *evaluation procedure* (E_i) available internationally and creating a *report* that results from it. A full-scale data model that supports the running example is available from [RL]. The running example components placed within the software architecture are modeled as an EJB application, and implemented within the J2EE, which is available from [MI].

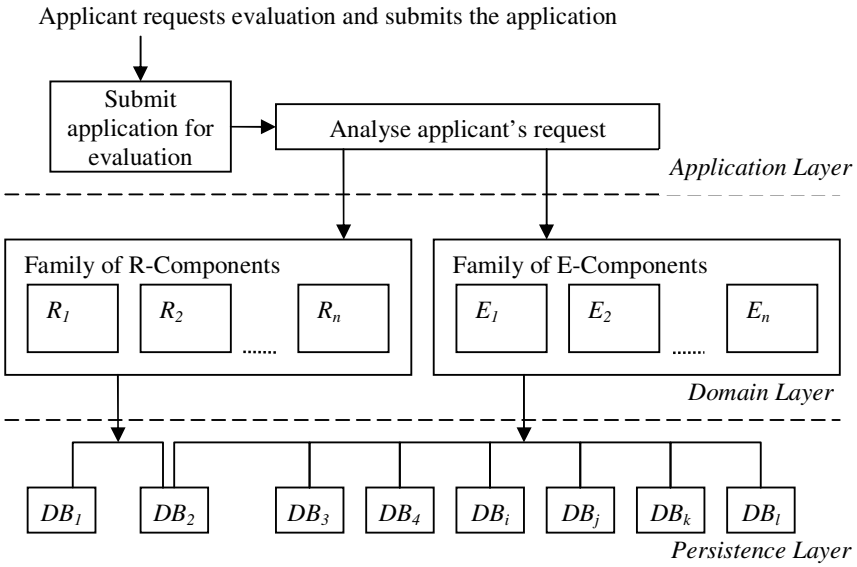

Figure 1: Component based reference architecture for automated evaluation of medicinal products

3 Problem Formulation and Related Work

In this paper we look at the problem of automation of medicinal products evaluation from a different perspective. Our experiences from the implementation of the running example have raised two issues:

(a) The deployment of components from our generic architectural model requires a component technology, which in turn dictates the design and implementation of the running example. In other words, the chosen component technology communication infrastructure is embedded within our example components, which compromises their independence.

(b) The complexity of the problem domain should gear us towards acquiring COTS components, which may alleviate the implementation of the proposed software architectural model and address the issue (a) above. Could we claim that some of the components we have generated in this problem domain can become COTS component and if yes, which characteristics should they have?

The aim of this paper is to address the (b)

We are not aware of any other work involving both COTS components and the problem of automation of medicinal products evaluations. We can only point towards the compatibility of our generic architecture with the existing distributed models in healthcare IS. Bringing new applications such as *Reporting on Adverse Drug Reactions* or *Updating Electronic Medical Records* will take our architectural model closer to the Distributed Healthcare Environment (DHE) middleware solutions for common health-care specific services, which has been formalised in European Health Information Systems Architecture (HISA) (Blobel, 2000). Our components from the domain layer (and some from the application layer) can find a place within the CEN middleware (Ferrara, 1998) of common services. A Health Level Seven (HL7), found at (http://www.hl7.org), an ANSI accredited US health industry communication (messaging) has extended the protocol for exchange of health-care information towards data from *master files, clinical trials* and *adverse drugs*. In version 3 they go beyond messaging. The Clinical Context Management Specification, XML Encoding for Version 2 and the Clinical Document Architecture are products that complements HL7's messaging standards. It remains to be seen if we can view HL7 as realising our domain specific messages and interfaces for information interchange, which can support communication between different component layers of our proposed architecture (between different applications and between applications and underlying infrastructure).

4 Creating COTS Components

The potential COTS components, which can be used when automating medicinal product evaluations, have emerged from the final model of our application, i.e. after the decision on component technology was made and after a few design patterns that suited any similar application from the same problem domain were generated. Therefore in this section we describe the whole procedure of modelling the application, generating design patterns and extracting potential COTS components.

4.1 Designing the Application Components

When designing our application we adopted the four principles in the following order:

(I) Choosing an adequate component technology
(II) Adhering to the layering principle and constraints from the architectural model
(III) Applying the MVC design model as suggested by the J2EE framework

(IV) Generating our own design patterns dictated by the functionality of the problem domain and the MVC design model.

In this section we describe (I)-(III) and leave (IV) for section 4.2.

(I) Choosing an adequate component technology:

Our analysis of component platforms of frameworks has short-listed the following three options suitable for the problem domain and the architectural model:

1. the EJB and the J2EE platform
2. a framework aligned within the CORBAmed standards from (http://www.omg.org/corbamed/),
3. an Internet based n-tier architecture adopting a CORBA middleware layer such as the Artemis architecture from (Jagannathan, 2001).

The complexity of the CORBAmed framework and experimental status of the Artemis prototype lead us towards the J2EE platform. This does not mean that we will not use Artemis or CORBAmed in future. . We have been geared towards J2EE because of our previous positive experiences of designing an application for database interoperability as an EJB application [IASTED papers and ITI paper]. Our decision to use EJBs has also been based on the fact that (a) EJBs are portable between various vendor implementations of J2EE, (b) EJB standard has been adopted by a number of vendors in order to provide EJB-compliant servers EJB and (c) EJB containers could shield us from component implementation complexities. More discussion on the topic is available at [xxxxx].

(II) Adhering to the layering principle and constraints from the architectural model

Our layered architecture conforms to [39] where layers are *"allowed to use"* public facilities of the nearest lower level (the usage of layers flows downwards). Our layering principle aims to achieve a certain degree of flexibility, re-usability and extensibility of the architectural solution:

- We separate components into layers according to their specificity within the application. However, the core layer (domain specific) components push away application specific requirements from generic functionality of data sources/computing platforms, making systems more adaptable to changes.
- The content of a particular component may be decided upon which layer it is appropriate to reside, i.e. knowing the layer in which the component resides, we know which services it offers.
- We can *extend families of domain specific components* if functionality requires, without affecting existing components in the same and adjacent layers. Furthermore, we may *generate in advance domain specific components* to suit new requirements/applications.

Consequently, our design model should distinguish between components placed in different layers, i.e. we should know at any time which 'type' of component is included in which part of the design model.

(III) Applying the MVC design model as suggested by the J2EE framework

The components from the application specific layer are represented by JSP and servlets in order to display and obtain information form the user. Srevlets also implement workflow and session management. Components from the application layer accept a user input, analyse it, make invocations to the EJB components, and issue a response to a user. We use servlets as the common entry point into the application. It is supported by a controller role given to servlets in JSP/servlet/EJB scenarios of the MVC [reference an adequate source for MVC] design model. This model enforces a separation of Model (EntityBenas or/and JavaBeans), View (any HTML file and/or JSP) and Controller (servlets and SessionBeans) aspects.

WE give some examples of using the MVC model when accessing DB records and when performing the functionality of evaluating medicinal product.

Example 1:

Our servlets allow access to DB elements either directly through EntityBeans or use SessionBeans as intermediary to retrieve records. In our design we follow the view to:

- use EntityBeans if the result of retrieval is a single record
- use SessionBeans and EntitiBean when multiple records are to be retrieved. Such a SessionBean is named as 'Look-up'.

Figure 2 shows an example of "look-up" SessionBeans for viewing all reports associated with a given application: View.Servlet needs <<LookupGetReportHistory.SessionBean>> for retrieving all reports associated to a particular application;

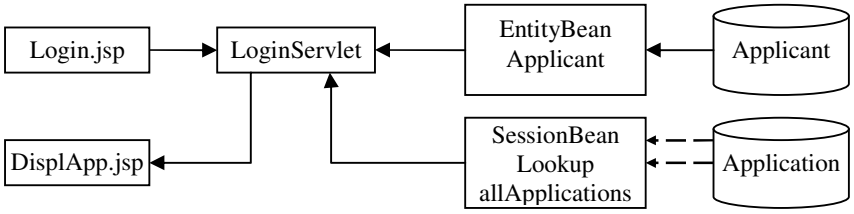

Figure 2:

Example 2:

Our Servlets also control the application functionality. They control *submission* of new applications and *evaluation* of submitted applications.

In both cases we need an EntityBean such as <<EvaluationEntityBean>> or <<RuleEntityBean>>, which retrieves a stored rule for checking the submission and a chosen evaluation procedure for evaluating the application. These EntityBeans are PLUGS-IN into SessionBeans that perform submission and evaluation, and they may be available locally or at remote nodes.

Figure 3 shows the example of evaluating submitted application, controlled by EvaluationServlet which delegates <<EvaluatingSessionBean>> to carry out the evaluation. We need <<EvaluationEntityBean>>, which retrieves a chosen evaluation procedure for evaluating the application and plugs it into <<EvaluatingSession-Bean>>

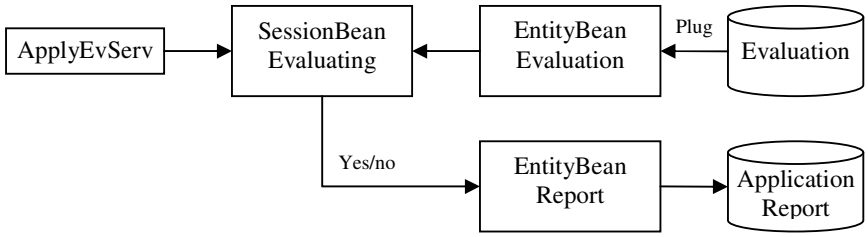

Figure 3:

Example 3:

EntityBeans are also used when updating and entering records in the DB. Updating of DB records is a consequence of performing evaluations and conformity to 'rules' when doing submissions. In our prototype we assume that there is only ONE 'rule-checking' procedure, which is prescribed by a local evaluation body (see wrkflow(a) in section 2). Consequently, there is no need to use a look-up Session-Bean for retrieving rule-checking as opposed to retrieval of evaluation procedures (<<Look-upEvaluations.SessionBean>>), where we can chose amongst all of them available locally and globally. This is shown in Figure 4.

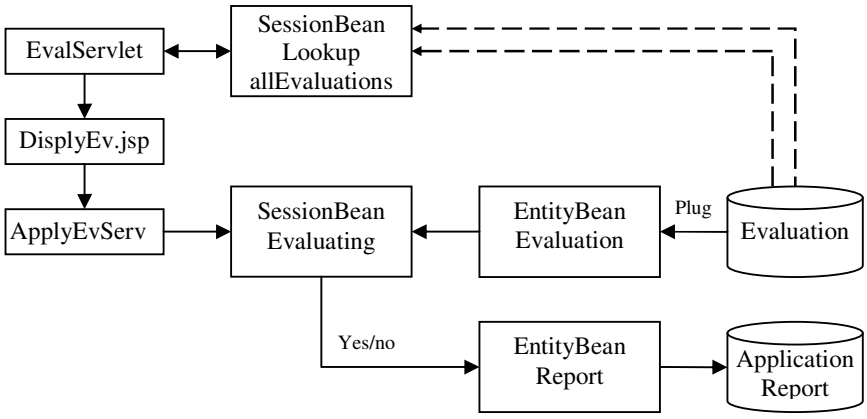

Figure 4

4.2. Creating Design Patterns

The *Strategy* pattern from [Gamma] is used within the core layer of Figure 1 when generating a family of components that implement the functionality of *checking adherence to submission rules* and *evaluating* submitted applications. These families of R_i and E_i components provide different implementations of the same behavior, where the user's request (and user's understanding of the problem[1]) decides the most suitable implementation. This pattern helps us to vary one part of our architectural structure independently to some other parts, making our system more robust to change, addressing reusability and achieving extensibility. We argue that:

1) 'submission rules' and 'evaluation' procedures' as parts of software that are likely to change in future (e.g. to be extended, optimised or to be changed completely) are isolated from the rest of the system;
2) we may define as many different variants of the same 'submission rules' or 'evaluation procedures' as possible, i.e. a family of 'rules' and 'evaluations'. This also means that we may generate new 'checking rules' and 'evaluation procedures' through previous experiences, new legislations/ICH and similar;
3) the user of the system chooses the most suitable combination of submission rules' and 'evaluation procedure', i.e. different tactics according to trade-offs when evaluating applications (N.B. the user is aware of different rules/procedures – this is a requirement of the *Strategy* pattern);

We also follow some J2EE patterns reference J2EE patterns book]. Figure 5 shows a typical "front controller": our ChoiceButton.Servlet controls the whole application by allowing the user to click either Submission, View or Evaluation buttons!

[1] This user decides on which submission rules and evaluation procedure are to be applied.

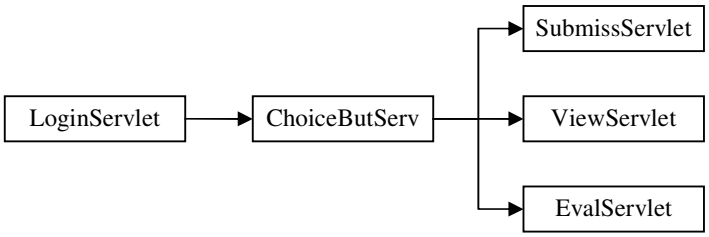

Figure 5

Our design patterns 9 (**I have to think about them**!) are suited any similar application from the same problem domain were generated. Therefore in this section we describe the whole procedure of modelling the application, generating design patterns and extracting potential COTS components.

4.1 Candidates for COTS Components

To be done – short section

5 Characteristics of COTS Components

6 Conclusions