Design patterns for automation of marketing authorisations in pharmaceutical industry.

Stephen Williams
Radmila Juric
Peter Milligan

School of Informatics

Design Patterns for Automation of Marketing Authorisations in Pharmaceutical Industry

Stephen Williams, Radmila Juric, Peter Milligan*
Cavendish School of Computer Science, University of Westminster, London, UK
**Enterprise Architecture, DCSIT, GlaxoSmithKline, London, UK*
williast@wmin.ac.uk, juricr@wmin.ac.uk, pete.s.milligan@gsk.com

Abstract. This paper reports on the experiences of using existing and creating new design patterns when deploying layered and component based software architecture that automates procedures for marketing authorization. We use the strategy pattern within the generic architecture and deploy the architectural components with the Model-View-Controller (MVC) and front controller patterns. Three domain specific patterns have been created and named as: look-up, submission and evaluation. We advocate that our combination of general and domain specific patterns (i) facilitate the design of distributed software applications, (ii) can be reused in any problem domain where workflows similar to submission and evaluations of application licenses occur, and (iii) comprise commercial-off-the-shelf (COTS) components that fit within our software architecture.

Keywords. Design patterns, marketing authorizations, COTS components, EJB.

1. Introduction

Component based software engineering has become a 'sine qua non' in software development, aiming to address its complexity, increase its productivity and decrease the development costs [20]. A development of software architecture contributes towards flexible and secure design of software systems [2]. We often describe software architectures in terms of components, interactions between them and patterns that guide a composition of components into systems. Design patterns [7] add more towards software development by representing knowledge, experiences and solutions for a class of reoccurring design problems, hence achieving higher software reusability and flexibility. Design patterns are often given in the form of structured descriptions of solutions for problems targeted by patterns. Precise design pattern specifications lead towards pattern based software development [4,18], where we can build software solutions from pattern specifications and systematically assemble them into a design [6]. However, informal description of patterns proved to be effective when communicating them to a wider audience, which could be underpinned by extraction of pattern solutions within existing software designs.

In this paper we report on experiences of using existing and creating new design patterns when deploying layered and component based software architecture that automates procedures for marketing authorisations of medicinal product licences. We do not formally specify our design patterns, but give examples and description of their role in our design. We also comment on their applicability across problem domains where workflows similar to marketing authorisations in the pharmaceutical industry occur. Our design patterns were extracted and enhanced after initial deployment of the architectural components took place. The choice of the J2EE technology has influenced our software design decisions and design patterns. Any discussion on the suitability of EJB [http://java.sun.com] and its impact to our design decisions, is outside the scope of this paper, but is available in our other publications [12,13,14].

We start with a layered and component based software architecture that automates procedures for marketing authorisations of medicinal product licences across the world, detailed in [10]. Please note that the 'marketing authorisation of medicinal product licences' is the procedure out of which a licence may be granted. In this paper we refer to this as 'marketing authorisation'. When deploying architectural example components we identify three kinds of design patterns that facilitate the design and implementation. We use the strategy pattern from [7], which is built into our software architecture. The MVC and front controller patterns [1] have been applied on the architectural example components after the

decision on using the EJB technology had been made. A set of our own design patterns, named as *look-up, submission* and *evaluation,* are problem domain specific. They have been created when deploying architectural components and implementing their functionality. We advocate that such a combination of general and domain specific patterns (i) facilitate the design of our distributed software application, (ii) can be reused in any problem domain where workflows similar to submission and evaluations of application licences occur, and (iii) comprise COTS components that fit within our architecture.

Section 2 details the related background and outlines our previous work on the automation of the marketing authorisations. Section 3 describes each pattern and emphasises the creation of our own patterns in the process of deploying a component-based model for the automation of marketing authorisations. In section 4 we comment on the role of COTS components within our design patterns. We conclude in section 5.

2. Related Background

Marketing authorisation procedures are very important tasks undertaken by government health departments and their regulatory authorities in every country in the world. However, each country has its own regulatory systems and marketing authorisation procedures, which represents a serious drawback for their efficient local and worldwide registration. The automation of such procedures in terms of adequate software support is a critical task that can improve the efficiency of regulatory authorities and interoperation of regulatory systems across the world.

We have analysed the local needs of regulatory authorities and have extracted their common practices across the world, which is essential if any interoperation between regulatory systems is to take place [8,9]. A software solution that automates such evaluation practices is a large-scale distributed application that requires sharing of data and processes associated with evaluations. In Fig. 1 we show the generic, layered and component based architectural model that allows automation of marketing authorizations. Each regulatory authority may apply their own authorization or any other available internationally. We define the generic marketing authorization that illustrates our software architecture from Fig. 1 as two workflows:

(i) *submission* of a licensing application for a marketing authorisation under local regulatory authority rules ($\boldsymbol{R_i}$),
(ii) *evaluation* of a successfully submitted licensing application, under an evaluation procedure and its rules $\boldsymbol{D(E_i)}$ available locally/internationally.

Details on both workflows and their operating environments are available at [14].

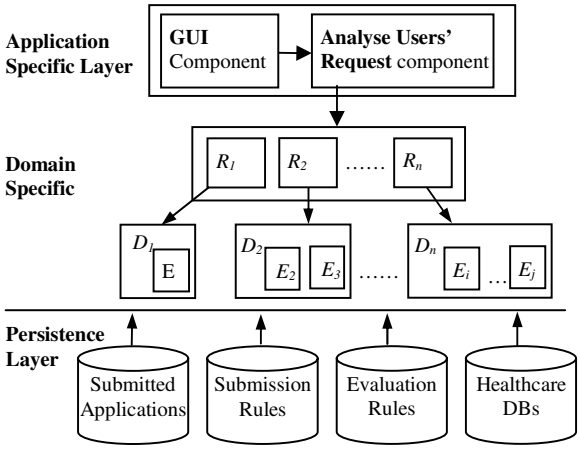

Figure 1: The Generic Software Architecture

The *application lay*er provides the basic GUI functionality and controls the interaction between users and any other layers within the system. This includes the right choice of $\boldsymbol{R_i}$ and $\boldsymbol{D_i(E_i)}$ components involved in a particular submission and evaluation of a medicinal product. The *domain layer* consists of two families of components: the $\boldsymbol{R_i}$ family 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 $\boldsymbol{D_i(E_i)}$ family contains all available *evaluation procedures* $\boldsymbol{E_i}$ that originate in different regulatory authorities and which can be applied to any submitted application (after the application has conformed to a set of submission rules in $\boldsymbol{R_i}$). With $\boldsymbol{D_i}$ we denote that we chose any combination of evaluation procedures $\boldsymbol{E_i}$, which are relevant for a particular $\boldsymbol{R_i}$, and of interest for a particular country/regulatory authority. Components from the *domain layer* use various data repositories stored within components of the *persistence layer*, where data on submissions and evaluations are kept. Our *persistence and domain layers* can be seen as a common repository of data and processes, where various applicants, such as pharmaceutical companies, regulatory authorities and hospitals,

can share data and services defined in our component architecture.

We have implemented a prototype, where example components from Fig. 1, are modelled as an EJB application, using Studio Enterprise 7 [13]. A full data-model that supports the example has been generated in ORACLE 8i [19]. We chose the J2EE platform for our implementation, because of our positive experiences of implementing software architecture for interoperable databases as an EJB application [12].

We are not aware of any work involving component-based solutions and design patterns when automating marketing authorisations. Our work on designing and implementing software architecture that addresses workflows (i) and (ii) is unique because each country uses local rules $\boldsymbol{R_i}$ for submission, and any available international evaluation procedure $\boldsymbol{D(E_i)}$. Thus, no specific design patterns have been available for and applicable to this problem. However, we are reusing the strategy pattern [7] for the same purpose and in the similar context as in our works on architectures for interoperable databases [11].

3. Design Patterns

We use existing design patterns and a set of our own that facilitate the design and implementation of our architectural components. The Strategy pattern [7] is built into our software architecture. The MVC and front controller patterns [1] were applied after the decision on using the EJB technology had been made. A set of our own design patterns is problem domain specific. It contains patterns called *look-up, submission* and *evaluation* that have been created when deploying our architectural components.

3.1 The Strategy Pattern

The *Strategy* pattern from [7] is used within the domain specific layer of Fig. 1 when generating $\boldsymbol{R_i}$ and $\boldsymbol{D(E_i)}$ components. They implement the functionality of the two workflows given in (i) and (ii) from section 2: submissions and evaluation of licensing applications. These families of $\boldsymbol{R_i}$ and $\boldsymbol{D(E_i)}$ components provide different implementations of the same behaviour, where the user's request (and user's understanding of the problem) decides the most suitable implementation or combination of $\boldsymbol{R_i}$ and $\boldsymbol{D(E_i)}$ components. This pattern helps 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 (for instance to be extended or optimised) are isolated from the rest of the system;
2) we may define as many variants of the same submission rules or evaluation procedures as possible, i.e. a family of submission rules and evaluations. We may generate new submission rules and evaluation through previous experiences, new legislations etc.
3) the user of the system chooses the most suitable combination of submission rules and evaluation procedure (N.B. the user is aware of different rules/procedures – this is a requirement of the *Strategy* pattern);

3.2 Applying the MVC Pattern

The components from the application specific layer of Fig. 1 are represented by JSP and Servlets in order to 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 the JSP/Servlet/EJB scenarios of the MVC pattern. It enforces a separation of Model (Entity Beans or/and JavaBeans), View (any HTML file and/or JSP) and Controller (Servlets and Session Beans) aspects. We do not show an example of the MVC pattern explicitly, but it can be found within Figs. 3, 4 and 5 where the separation of View and Controller is emphasised. Our Servlets implement workflows (i) and (ii) as in Figs. 4 and 5. This means that they control the flow of the application and do not engage in any business logic and do not control any of its data.

3.3. Using a J2EE Front Controller

We also follow the J2EE patterns [1]. Fig. 2 shows the front controller: ChoiceButServlet controls the whole application by allowing the user to click Submission, View or Evaluation buttons in order to carry out any of these functionalities.

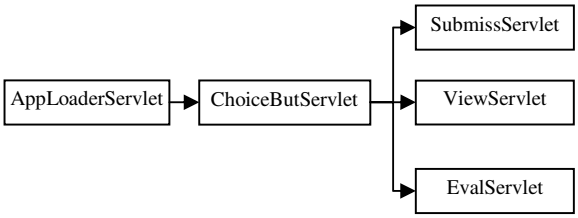

Figure 2: The Front Controller

3.4.Creating our own Design Patterns

We have already mentioned that our Servlets control the application functionality. They control *submission* of new applications through $\boldsymbol{R_i}$ and *evaluation* of submitted applications through $\boldsymbol{D_i(E_i)}$ components, which are the two workflows (i) and (ii) from section 2. In both cases we need an Entity Bean, which retrieves a stored rule for checking the submission and a chosen evaluation procedure for evaluating the submission (<<EvaluationEntityBean>> or <<RuleEntityBean>>). They both PLUG-IN to Session Beans that perform rule checking for submissions and evaluations, which may be available locally or at remote nodes.

Our Servlets also allow access to DB elements either directly through Entity Beans or using Session Beans as an intermediary. In our design we decided to use:

a) An Entity Bean, if the result of retrieval is a single record
b) A Session and an Entity Bean, if multiple records are to be retrieved. Such a Session Bean is named as 'Look-up'.

Thus we have discovered patterns that mirror the way we compose our components.

3.4.1 Look-up Pattern

To illustrate the look-up pattern, we give an example of retrieving all submitted applications for an applicant. Each applicant can have more than one licensing application within the system. If we want to retrieve all of them, we have to use Session Bean and Entity Bean as in b) from 3.4.

Fig. 3 shows an example of a "look-up" Session Bean for retrieving all submitted applications for a given applicant. For retrieving all applications, Login.Servlet delegates <<Look-upAllApplications.SessionBean>>, hence the double broken arrow between the Session Bean and Application database.

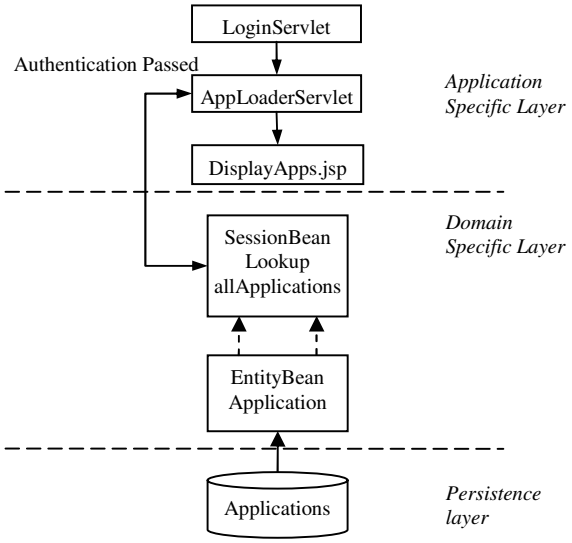

Figure 3: Look-up pattern

3.4.2. Submission Patterns

To illustrate the submission pattern we give an example of plugging a checking rule into a Session Bean. I.e. each licensing application must be submitted according to the submission rules R_i, which are stored within the Submission Rules database. If the submission does not adhere to the submission rules, a licensing application cannot be created within the system.

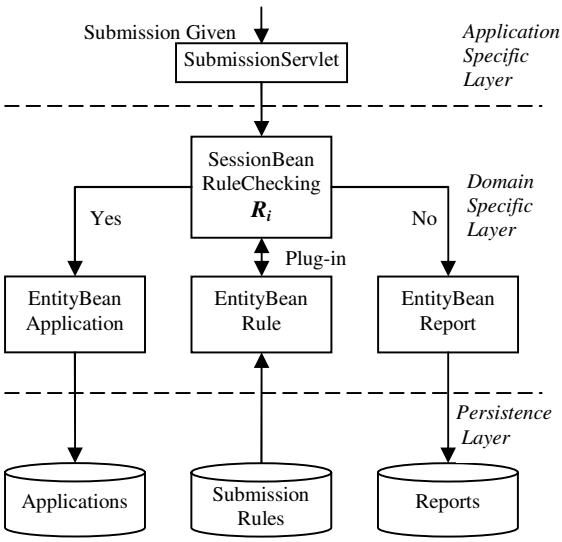

Figure 4: Submission Pattern.

Fig. 4 shows the submission of an application, which is controlled by the SubmissionServlet. It delegates <<RuleCheckingSessionBean>> to carry out the automatic submissions, i.e. checking the adherence to submission rules $\boldsymbol{R_i}$. An Entity Bean <<RuleEntityBean>> retrieves a

rule for checking the submission, which is stored within Submission Rules database and plugs it into <<RuleCheckingSessionBean>>. After checking the adherence to submission rules R_i, <<RuleCheckingSessionBean>> either creates an application with <<ApplicationEntityBean>> if the results of the checking are positive (Yes), or creates a report on unsuccessful submission with <<ReportEntityBean>> if the results of checking are negative (No).

3.4.3. Evaluation Pattern

To illustrate the evaluation pattern we give an example of executing an evaluation procedure using the $D_i(E_i)$ component which is applied to successfully submitted licensing applications. However, all available evaluation procedures have to be retrieved and displayed first, before choosing the most appropriate one.

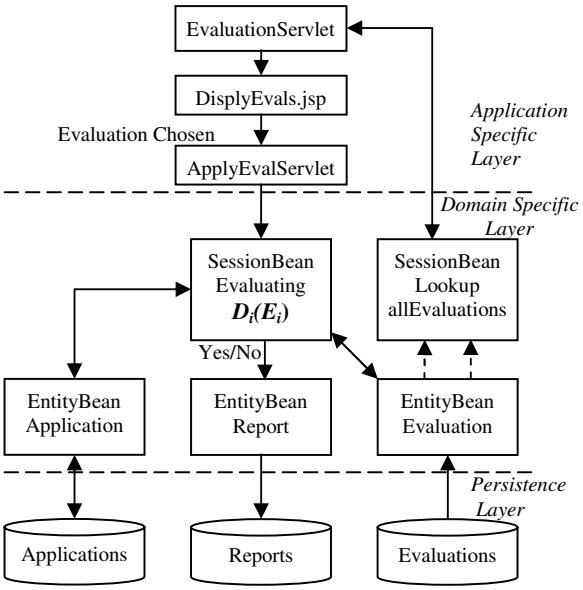

Figure 5: Evaluation Pattern

In Fig. 5 the evaluation of a submitted application is controlled by ApplyEvalServlet, which delegates <<SessionBeanEvaluating>>, i.e. $D_i(E_i)$ to carry out the evaluation. However, before the evaluation starts EvaluationServlet uses a look-up session bean called <<SessionBeanLook-upAllEvaluations>> for retrieving all evaluation procedures available locally or globally, which are stored within the persistence layer. After displaying them through DisplyEvals.jsp we chose one suitable evaluation procedure. We use an entity bean <<EntityBeanEvaluation>> to retrieve a chosen evaluation procedure from an adequate persistence. The <<EntityBeanEvaluation>> PLUGS into <<SessionBeanEvaluating>> $D_i(E_i)$ that perform evaluations. The result is stored using the <<ReportEntityBean>>. The evaluation pattern comprises look-up pattern: there may be more than one evaluation procedure available for each successfully submitted licensing application. Hence the need for the look-up pattern to retrieve them.

4. Design Patterns and COTS

The set of R_i and $D_i(E_i)$ components can be extendible, standardized and dynamically generated. They are candidates for COTS components: they represent an implementation of certain functionality, which can serve a family of related applications [15]. In our example we could use R_i and $D_i(E_i)$ components when submitting and evaluating any licensing application in any country in the world, i.e. according to any evaluation procedure. However, our R_i and $D_i(E_i)$ components could also be applied to any other problem domain where workflows similar to (i) and (ii) from section 2 take place (see our future works).

The R_i and $D_i(E_i)$ components operate on the principle of 'plugging-in' submission rules R_i or evaluation procedures $D_i(E_i)$ – both stored in a persistent data store - into Session Beans that implement these functionalities, which is shown in both patterns from Figs. 4 and 5. Therefore, the programming code stored within our <<SessionBeanEvaluating>> and <<SessionBeanRuleChecking>> could remain the same for a variety of evaluations (and submissions). What changes, are the submission rules and evaluation procedures stored within our Evaluation and Rule databases, which are plugged into an adequate Session Beans using EntityBeans [13]. Having COTS components, as part of the design patterns is an important feature: it facilitates design patterns reusability and addresses COTS components interoperability and dependability on component platforms [5].

5. Conclusions and Future Works

This paper reports on our experiences of using existing and creating new design patterns when deploying layered and component based software architecture that automates procedures for marketing authorisations. Some of the patterns are general and dictated by the use of component technology, and some of them are problem

domain specific. All design patterns contributed to the final design decisions, they are reusable across a family of related application and can accommodate COTS components.

We currently use a similar software architecture and our design patterns in two problem domains: (a) an automation of visa application submissions and their evaluations at the UK Home Office, and (b) the availability of undergraduate places and entry requirements across the universities in the UK and abroad. We are also developing a pattern language [3] that assists in the development of distributed component applications and enhances the reusability of business components as in [16]. We also plan to see how the set of $\boldsymbol{R_i}$ and $\boldsymbol{D_i(E_i)}$ components with design patterns affects design decisions, if we place them within frameworks that manage components' dependencies when assembling them into an application [17].

6. References

1. Alur D., Crupi J., Malks D. (2003) Core J2EE Patterns, 2nd edition, Prentice Hall
2. Bass L., P. Clements, R. Kazman (1998), Software Architecture in Practice, Addison Wesley
3. Brown K., Eskelin F., Pryce N. (1999) A Mini-Pattern Language for Distributed Design, in proceedings of the PLoP conf., pp
4. Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *A System of Patterns: Pattern-Oriented Software Architecture.* Wiley, 1996.
5. Chiang C.C.(2002), Development of Reusable Components through the Use of Adaptors, Proc. of the 36th Hawaii Int. Conf. on System Sciences (HICSS), IEEE, 2002.
6. France, R., S. Ghosh, E. Song and D.-K. Kim, A Metamodelling Approach to Pattern-Based Model Refactoring, *IEEE Software,* vol. 20, no. 5, Sept./Oct. 2003.
7. Gamma E, Helm R, Johnson R and Vissides J (1995) Design Patterns, Addison-Wesley Professional; 1st edition (January 15, 1995)
8. Juric, R, and J. Juric (1999) The Application of the UML to the Modelling of Automated Support of Evaluating Medicinal Products across Different Regulatory Requirements, in D. Kalpic, V. Hljuz-Dobric (eds.) *Proc. of the 21st Int. Conference on ITI'99, Pula Croatia,* pp. 283-291, ISSN 1330-1012.
9. Juric R. J. Juric (2000) Applying The UML Modelling Elements in Complex Business Environment, in M.M. Tanik and A. Ertas (eds.) Proc. of *5th IDPT 2000 Conference,* June, Dallas, Texas, US, ISSN 1090 – 9389
10. Juric R., and J. Juric (2002) Applying Component Based Modelling in the Process of Evaluation of Medicinal Products**,** in the proc. of the *IDPT-2002 conference*, Pasadena, CA, US, ISSN 1090 – 9389
11. R. Juric, J. Kuljis, P. Paul (2004), Software Architecture Style for Interoperable Databases, *Proceedings of the 26th Int. Conf. on ITI '04*, Croatia, 2004
12. R. Juric R. and LJ. Beus-Dukic (2005), COTS components and DB Interoperability, in Proc. of the 4th International Conference COTS-Based Software Systems ICCBSS 2005, Spain, LNCS 3412, pp. 77-89.
13. R Juric, S. Williams, L. Slevin, R. Shojanori S. Courtenage, P. Milligan (2005) Component Based Automation of Marketing Authorisations, submitted to the Journal of Healthcare Informatics.
14. R Juric, L. Slevin, R. Shojanori S. S. Williams (2005a) Software Support in Automation of Medicinal Product Evaluations, to appear in proc. of the ICMCC event, Hague, 1-3 June 2005.
15. Juric, R., and Williams, S., (2005), Experiences of Generating COTS Components when Automating Medicinal Product Evaluations, submitted to *The 7th Int. Conference on Software Engineering and Knowledge Engineering, SEKE '05.*
16. Neill C.J. and Bharminder G. (2003) Refactoring Reusable Business Components, in ITPro, IEEE, Jan/Feb 2003, pp 33-38
17. Northcott M. and M. Vigder (2005) Managing Dependencies between Software Products, in in Proc. of the 4th International Conference COTS-Based Software Systems ICCBSS 2005, Bilbao, Spain, LNCS 3412, Springer-Verlag, pp. 201-211.
18. Rising, L. (1998) The patterns Handbook: Techniques, Strategies, and Applications, SIGS Books, Cambridge University Press
19. Slevin, L., R. Shojanori and R. Juric (2005), A DB Environment for Automating Regulatory Affairs in the Pharmaceutical Industry, to appear in Proc. of the 8th Int. Conference on Integrated Design and Process Technology (IDPT), Beijing, China
20. Szypersky C., (2002) "Component Software-Beyond Object Oriented Programming", Addison Wesley.