Deriving Software Architectures for CRUD Applications: The FPL Tower Interface Case Study

Atif Mashkoor, João M. Fernandes
Departamento de Informática / CCTC
Escola de Engenharia, Universidade do Minho
Braga, Portugal
{atif,jmf}@di.uminho.pt

ABSTRACT
The main aim of this paper is to present how to derive logical software architectures for CRUD (Create, Read, Update and Delete) applications using a specific technique called 4SRS. In this technique, a component diagram, which is obtained through transformations of use cases, is used to represent the logical software architecture. To show that the 4SRS technique, which was initially devised for behavior-intensive reactive systems, is also effective and gives seamless results for other software domains, it is being experimented on data processing systems, which typically follow a CRUD pattern. For demonstration purposes, the FPL tower interface system, which is responsible for communication between air traffic control operators and flight data processing system on airports of Portugal, has been used as a case study.

KEYWORDS
Software engineering, software design, software architecture, use cases, component diagram, UML

1. INTRODUCTION
According to some authors [3, 10], the transformation of requirement specifications into software architecture is one of the most complex activities of software development. One of such architectural transformation techniques is the 4-Step Rule Set (4SRS) [5, 6] that employs successive transformations of use cases to obtain a logical software architecture that satisfies the elicited user requirements. It provides a comprehensive set of guidelines divided into four steps that helps developers to obtain an initial architecture for the software system in a consistent, coherent and systematic way. The iterative nature of the approach ensures that the derived architecture reflects the user requirements as seamless as possible.

Initially, the 4SRS technique was proposed for behavior-intensive systems and has been used for reactive embedded systems for numerous times [4, 7, 8, 9]. This approach seems also applicable to other domains, such as CRUD applications, which are data-centric in nature rather than behavior-intensive.

The acronym CRUD stands for Create, Read, Update and Delete. These four operations stay at the core of CRUD applications and play a pivotal role in the system. CRUD applications, which are data-oriented, work with such data modules that are retrieved, modified, updated and sent back to applications for persistence. Data processing systems can be seen as typical CRUD applications due to their strong orientation towards data. Systems responsible for data processing provide different mechanisms of data conversion into information through handling, sorting, and computation of data in compliance with defined protocols.

The purpose of this paper is to demonstrate that the 4SRS technique can also be used to derive software architectures for CRUD applications as successfully as for reactive embedded systems. To support this hypothesis a case study, the FPL tower interface system, has been used. This system is responsible for the communication between air traffic control operators and flight data processing system on airports of Portugal. It is a data processing system and can be referred to as a typical CRUD application. We apply the 4SRS technique to this system and derive its logical software architecture through successive transformations of uses cases.

The paper is structured as follows: section 2 gives an overview of the 4SRS technique, section 3 introduces the case study, section 4 illustrates the application and execution of the 4SRS technique on the case study, and finally section 5 draws some conclusions and suggests directions for future work.

2. THE 4SRS TECHNIQUE
4SRS is a stepwise technique that transforms a use case diagram into a component diagram, to help software developers obtaining the final architectural design. The updated 4SRS technique, based on [11], presented here, is divided into four main steps and six micro-steps:

International Conference on Software Engineering Advances(ICSEA 2007)
0-7695-2937-2/07 $25.00 © 2007 IEEE

1. Component creation
2. Component elimination
   - 2i. Use case classification
   - 2ii. Local elimination
   - 2iii. Component description
   - 2iv. Component representation
   - 2v. Global elimination
   - 2vi. Component naming
3. Component packaging
4. Component association

The first step of the 4SRS technique creates three components for each use case i.e. *control, data* and *interface* components. Each component receives the reference of its respective use case appended with the suffix (c, d, i) that indicates the category of the component. However to deal with FPL tower interface system, which is a data- and transaction-centric system and holds only one shared repository of data for all components, we do not create data component in this first step; instead just *control* and *interface* components are created. The data component will eventually be incorporated into the resulting component diagram.

The main aim of the second step is to eliminate those components created in the first step that are not relevant for the architecture, i.e., they are not required to represent the functionality of their particular use cases. In the first micro-step of this step, the category of components is determined based on the textual description for each use case, which helps deciding which components to maintain and which ones to discard. In the second micro-step, the information gathered in first micro-step is used to discard the components that do not make sense in the problem domain. The next micro-step is used to describe the components, based on the textual descriptions of the original use case. In the fourth micro-step, the redundancy of the user requirements elicitation is eliminated and the missing requirements are added to the system. In this step, every component passes a so-called self-sustainability test to determine if the component can be represented by other component or not. In a positive case, it means that the representative component will not only represent its own system requirements but also the requirements of the represented components. Otherwise, it means that the component will only represent itself. Micro-step 2v is straightforward since it only consists of the application of the results of the last micro-step. The components that are represented by other components must be eliminated because their system requirements are now responsibility of the representative component. This micro-step is called "global elimination" due to its global awareness for generating a coherent and cohesive component model, from the point of view of system requirements. The last micro-step names the components. The name received by the component must reflect its role in the system, the use case which it comes from, and additionally it must reflect all the represented system requirements as well if it is representing another component.

The third step organizes and unifies the surviving and semantically consistent components into packages. These packages are the orchestration of similar components into groups to present the architecture at higher levels thus making it easier to understand.

The fourth step links the aggregates to specify the associations among the existing components. These associations, which work as connectors between these components, show provided or required functionality or any kind of dependency among these components.

The changes proposed by the upgraded version of the 4SRS technique, which is presented in this paper, over its successors are as follows: first, it replaces the word of object by component that is a more meaningful concept. Secondly, it optimizes the second step by eliminating the redundancy of naming the component. Originally the component used to be named in the first place and then renamed afterwards if it was changed. But now we only name it once its functionality is fully determined. We also change the name of the third step from "component packaging and aggregation" to just "component packaging". This is due to the fact that in the second step where components passes a so-called self-sustainability test, the aggregation already takes place at that time and in third step we just now pack them. To represent the derived architecture we now use a component diagram instead of an object diagram which, according to [2], is more meaningful and appropriate to represent software architectures.

3. THE CASE STUDY

The Flight Data Processing System (FDPS) is responsible for flight data processing in the Portuguese airspace including the control towers of Porto, Faro, Lisbon, and Funchal [12]. The Flight Plan (FPL) tower interface, the system under consideration for this case study, is responsible for interfacing between air traffic control operators and FDPS. Most of the data used by FPL is obtained from the Airport Operational System (AOS), Flight Data Section (FDS) and Environment. All these systems are connected to FDPS, through a middleware called BasicSystem that provides services for message handling, task management, buffer management, and memory management.

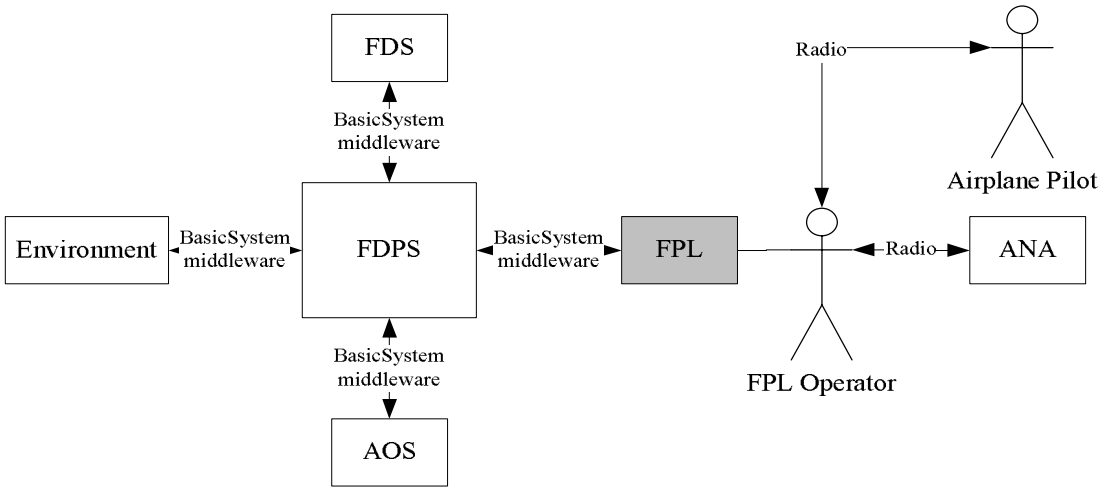

Figure 1: A big picture of the system

Further communication with the air control operators occurs by radio, namely with ANA Aeroportos de Portugal [1] and the aircraft pilots. Fig. 1 shows the overall structure of the system where FPL is included.

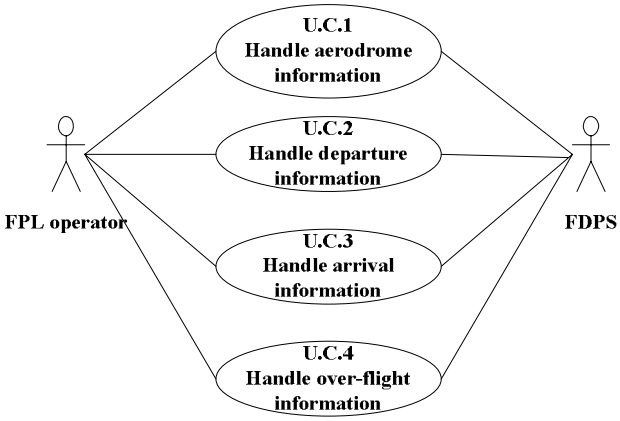

Figure 2: Top level use case diagram for FPL

This case study implements the mechanisms used by the tower operators for managing the tower related information such as monitoring local aerodrome data, and monitoring flight data relative to the arrival, departure and over-flights involving the local aerodrome. Fig. 2 shows the top level use case diagram for FPL tower interface which offers four main groups of functionalities. *U.C.1 handle aerodrome information* allows the operator to update the data of aerodrome. *U.C.2 handle departure information* is responsible for a list of flights departing from the aerodrome. *U.C.3 handle arrival information* is responsible for the flights arriving at the aerodrome. *U.C.4 handle over-flight information* manages the information of flights affecting the locally controlled airspace, without involving the aerodrome. For instance, flight information of aircraft inside the range of locally controlled airspace, neither arriving at nor departing from the aerodrome, shall be considered as *U.C.4*. Each use case can list, visualize and modify the parameters of the respective item. Thus, the considered case study clearly constitutes a CRUD application.

4. THE 4SRS TECHNIQUE EXECUTION

We identify *U.C.3* to be used as an example of a CRUD application, as all other use cases possess the same type of CRUD functionalities. Fig. 3 shows the expanded view of *U.C.3 handling arrival information*. *U.C.3.1 refresh arrival data* checks and fetches the latest updated arrival data from the FDPS. *U.C.3.2 confirm arrival data* asks

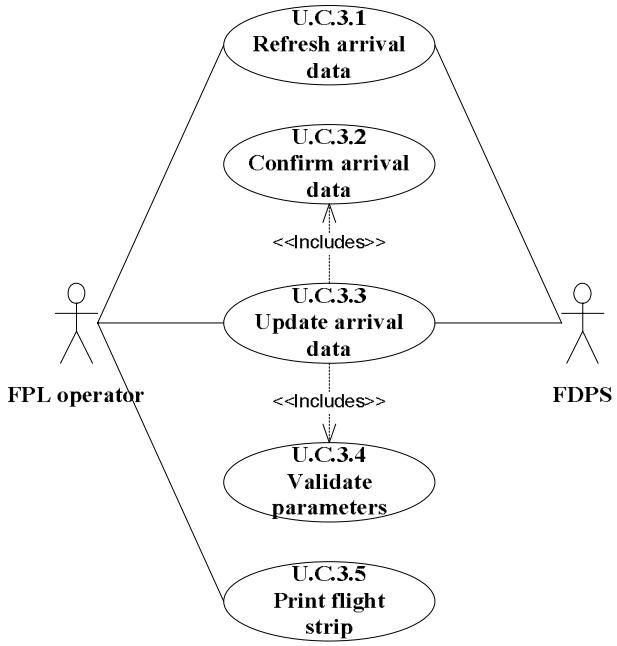

Figure 3: Expanded *handling arrival information* use case

Step 1	Step 2							Step 3	Step 4
Component creation	2i: Use case classification	2ii: Local elimination	2iii: Component description	2iv: Component representation		2v: Global elimination	2vi: Component naming	Component packing	Component linking
				is represented by	represents				
U.C. 3.1 Refresh arrival data	ci								
3.1c			Checks if the data has been refreshed	Itself			Data refreshing control	PI	3.1i
3.1d		X							
3.1i			UI	Itself			Refresh interface	PII	3.1c
U.C. 3.2 Confirm arrival data	i								
3.2c		X							
3.2d		X							
3.2i			UI	Itself			Arrival confirmation interface	PII	3.3i
U.C. 3.3 Update arrival data	ci								
3.3c			Update data in FDPS	Itself			Arrival update control	PI	3.3i
3.3d		X							
3.3i			UI	Itself			Arrival update interface	PII	3.2i, 3.3c, 3.4c
U.C. 3.4 Validate parameters	c								
3.4c			Validates the parameters before update	Itself			Arrival parameter validation control	PI	3.3i
3.4d		X							
3.4i		X							
U.C. 3.5 Print flight strip	ci								
3.5c			Printing code	Itself			Arrival strip printing control	PI	3.5i
3.5d		X							
3.5i			UI	Itself			Arrival strip printing interface	PII	3.5c

Table 1: Tabular representation of the 4SRS technique execution

the FPL operator regarding his/her certainty about a particular operation, for instance, if the operator wants to update particular flight data then a dialog appears for confirmation purposes. *U.C.3.3 update arrival data* is responsible for updating the data in FDPS. *U.C.3.4 validate parameters* makes sure that no data anomaly is created in the database during an update operation. *U.C.3.5 print flight strip* is dedicated to print paper strips about the flight information, which may be needed in case of system failure. We now apply the 4SRS technique step by step.

Step 1: Component creation

In this step, each use case is transformed into two components classified with one of the following categories: *control* or *interface*. In this step there is no need to take or validate any decision. As a result of the application of this step to the case study, the *control* and *interface* components are created for each use case. All created components are shown in the tabular representation of the 4SRS technique in table 1.

Step 2: Component elimination

This step is one of the most crucial steps of the 4SRS technique. The success of the whole technique is only assured provided that definitive system-level entities are identified during this step. The aim of this component elimination step is to decide which of the components created in the first step must be kept in the model. Those components that are not subsequently eliminated must fully represent the use case. This decision must take into account the whole system including the textual description of each use case, rather than each use case in isolation. Additionally this step allows the elimination of the redundant user requirements and also tries to fill the gap by discovering missing requirements. This step is further decomposed into several micro steps due to its complexity.

Micro-step 2i: Use case classification

This first micro-step classifies each use case to help on the transformation into components. It also gives some hints on how to categorize and connect use cases and their respective transformed components. For example, *U.C.3.1*

is classified as "ci", which means that both components (*control* and *interface*) are kept for this use case; *U.C.3.2* is classified as "i", meaning that only the *interface* component for this use case is maintained. Table 1 shows these decisions.

Micro-step 2ii: Local elimination
In this micro-step, we analyze if every component created in previous step makes sense or not to be kept in the architecture. If not, we simply eliminate it. Here, components {*3.1d, 3.2c, 3.2d, 3.3d, 3.4d, 3.4i, and 3.5d*} do not make sense in the problem domain, so they are eliminated. All data components are eliminated due to the fact that eventually one data repository replaces all of them. For the rest of the eliminated components, the decision can be justified by analyzing the textual descriptions of each affected use case. For instance, use case *U.C.3.4* is responsible for validating the parameters for the update process. This description originates only one control component, whose responsibility is validation. This leads to the inclusion of component {*3.4c*} into the component model.

Micro-step 2iii: Component description
In this step all components are described. The descriptions are based on the original use case textual descriptions. One example of a component description is the elicitation of component {*3.4c*}. Component {*3.4c*} is responsible for parameter validation and must provide data checks before updating data. Table 1 shows the abridged description of each component in the column for this micro-step.

Micro-step 2iv: Component representation
In this step, in order to avoid redundancy, it is checked for each component if it can be represented by other component. This means that the representative component can not only represent its own system requirements but also the requirements of the represented components as well. In this particular case, all components are self-maintainable and do not require any representation hence no representative component is produced.

Micro-step 2v: Global elimination
The components, which can be represented by other components, must be eliminated in this step to have a smooth design. Since no representative component has been produced in the previous micro-step, global elimination does not take place for this case study.

Micro-step 2vi: Component naming
Now the components must be named. The name received by the component reflects its role and originating use case. For instance, components {*3.1i*} and {*3.1c*} receive the name *refresh interface* and *data refreshing control* respectively. Table 1 includes the names of all components.

Step 3: Component packaging
In this step, we package those components that are somehow related to each other in any context. In this particular case study, the criteria to aggregate these components is the type of components, i.e., *control* components are packed together in package *PI* and *interface* components are packaged as *PII. S*ince only one data component exists in the system there is no need to pack it.

Step 4: Component association
The fourth and final step of the 4SRS technique introduces the links in the component model. The decisions to include the links are strongly based on the textual information of the use cases, but also use other available information, such as stereotypes. The association in this component diagram is shown with the help of dependencies, required interfaces, and provided interfaces. For instance, component {*3.1c*} is providing refreshed data to component {*3.1i*}. This step must be done carefully in order to avoid any design error.

After applying successive transformations and eventually linking the components, the resulted component diagram is obtained (fig. 4). The rectangles refer to components of the system, the dashed lines denote dependency, and required and provided interfaces are shown by socket and lollypop symbols respectively.

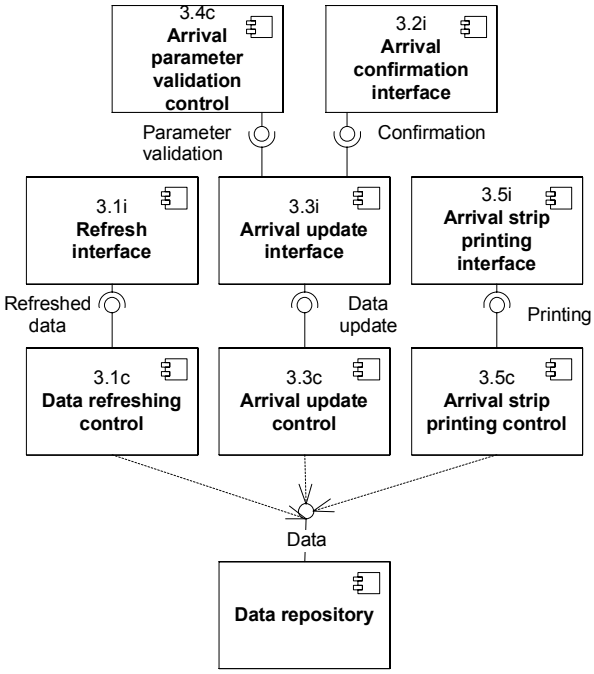

Figure 4: The resulted component diagram

Since we have come up with a logical architecture with three separate layers of interface, controls and data, this leads us to the three-tier architectural approach. In three-tier architectures functionality of the application is distributed across separate and independent layers of user interface, functional logic, and data storage and access. The newly created component diagram presents the initial and simple architectural reference of the system emerged from the transformations of requirement model. Further details can be added to the resulted architecture to acquire the final software architecture of the system.

After obtaining this new architectural model further characteristics of the application design can be modeled through different diagrams. For example, a class diagram can be used to define the static structure and state machine diagrams, activity diagrams and interaction diagrams can be used to model the behavioral characterization of the application. Creation of these artifacts is out of scope for both the 4SRS technique and this paper.

5. CONCLUSIONS AND FUTURE WORK

Obtaining software architecture from user requirements is a challenging and important task for software developers. Any misjudgment by software engineers in this process may introduce severe design flaws and inconsistencies. One approach to support this derivation is the 4SRS technique that helps in transforming a use case diagram into the corresponding logical software architecture in a systematic, coherent and seamless manner. The 4SRS technique has proved its value in behavior-intensive systems, such as reactive embedded software design, yet it was never leveraged to other types of software systems.

This paper applies the 4SRS technique to CRUD applications. To show this application the paper uses the FPL tower interface system, a data processing system, as a case study. The derived logical software architecture presents the typical component model of such kind of CRUD applications. The 4SRS technique also shows its usefulness by assuring the generation of a seamless specification of the architectural requirements. The resulted component diagram suggests that 4SRS can also be applied to CRUD applications with similar benefits as those encountered for reactive embedded systems. In particular, it facilitates the generation of three-tier architectures, which are a commonly accepted solution for data-centric software systems.

In the future, we intend to incorporate aspectual support to the 4SRS technique. We would like to evaluate how the technique can benefit from its combined application with Aspect Oriented Requirement Engineering (AORE) approaches to represent software architectures.

ACKNOWLEDGEMENTS

We thank Ana Moreira, João Araújo, André Marques, and Sérgio Agostinho for providing documents and support, and Paula Santos for her comments and suggestions with respect to the case study. This work was supported by the SOFTAS project (POSC/EIA/60189/2004).

REFERENCES

[1] ANA Aeroportos de Portugal SA, www.ana.pt, lastly accessed: Mar/2007

[2] S. Ambler, The Object Primer, 3rd Edition, Cambridge University Press, 2004

[3] J. Bosch, P. Molin, Software Architecture Design: Evaluation and Transformation, 7th IEEE Conf. on the Engineering of Computer-Based Systems (ECBS'99), pp. 4-10, IEEE CS Press, 1999

[4] A. Bragança, R. J. Machado, Adopting Computational Independent Models for Derivation of Architectural Requirements of Software Product Lines, 4th Int. Workshop on Model Based Methodologies for Pervasive and Embedded Software (MOMPES 2007), pp. 91-101, IEEE CS Press, 2007

[5] J. M. Fernandes, R. J. Machado, From Use Cases to Components: An Industrial Information Systems Case Study Analysis, 7th Int. Conf. on Component-Oriented Information Systems (OOIS '01), pp. 319-328, Springer-Verlag, 2001

[6] J. M. Fernandes, R. J. Machado, H. D. Santos, Modeling Industrial Embedded Systems with UML, 8th ACM/IEEE/IFIP Int. Workshop on Hardware/Software Codesign (CODES'2000), pp. 18-22, ACM Press, 2000

[7] J. M. Fernandes, R. J. Machado, P. Monteiro, H. Rodrigues, A Demonstration Case on the Transformation of Software Architectures for Mobile Applications, IFIP Conf. on Distributed and Parallel Embedded Systems (DIPES 2006), pp. 235-244, Springer-Verlag, 2006

[8] J. M. Fernandes, R. J. Machado, System-Level Component-Orientation in the Specification and Validation of Embedded Systems, 14th Symp. on Integrated Circuits and Systems Design (SBCCI 2001), pp. 8-13, IEEE CS Press, 2001

[9] J. M. Fernandes, R. J. Machado, P. Monteiro, H. Rodrigues, Refinement of Software Architectures by Recursive Model Transformations, 7th Int. Conf. on Product-Focused Software Process Improvement (PROFES 2006), pp. 422-428, Springer-Verlag, 2006

[10] H. Kaindl, Difficulties in the Transition from OO Analysis to Design, IEEE Software, vol. 16, nr. 5, pp. 94-102, 1999

[11] P. Monteiro, Model-based Transformations for Software Architectures: A pervasive application case study, Master thesis, Dept. Informatics, University of Minho, Portugal, 2005

[12] NAV Portugal, E.P.E., www.nav.pt, lastly accessed: Mar/2007