# Chapter 7

# System Design

## 7.1 Introduction

In this chapter, we present the architectural design of the SA Workbench software. As it was mentioned in the previous chapters, since during the initial stage of the project development we had identified all system's requirements and that those requirements were unlikely to change, we decided to use the Attribute Driven-Design (ADD) method to design the software architecture of this software-intensive system. The initial stage of ADD involves identification, categorization, and rating of the system's requirements. After this initial stage, the ADD process continues by a recursive decomposition of the system [7]. Due to the word limit of this dissertation thesis we do not describe the entire ADD process but rather we discuss some of the key aspects of the resulting software architecture design.

The system's design is presented from different perspectives which form a coherent description of the system's architecture. Each perspective is applicable to a specific group of software stakeholders since it presents system's attributes that are related to them. Such an approach has a number of advantages over the use of a single view model among which are that the resulting design is better understood by the stakeholders and it is easier to maintain since different perspectives are not mixed. The Open Group, which is a consortium for development of IT standards and certificates, suggests that, at the design stage, the architect chooses and develops a set of views such that could be understood by the stakeholders, enable the communication of the software architecture, and could be used for verification that the designed system addresses the system's requirements [6]. This suggestion implies that there is not a set of views applicable to every project but rather that the set of views selected or developed depends on the project's needs. For the purposes of this project we have selected the set of views proposed by Philippe Kruchten in their work called "Architectural Blueprints - The '4+1' View Model of Software Architecture" [46].

We have selected this set of views since the proposed views are well understood by the architectural stakeholders, due to its clear structure, and allow us to effectively communicate the software architecture. This view model includes five views namely, Logical, Process, Physical, Development, and Scenarios.

The software architecture design was developed prior the development of SA Workbench but postpartum it has been redrawn using its design tool. Each view is represented using the developed for the SA Workbench design tool notation.

## 7.2 Design Requirements

After identification of the system's requirements, following the ADD process, we have categorised and rated them. The requirements were categorised in the following categories: Design Constraints, Functional Requirements, and Quality Attribute Requirements and were rated in relation to their impact on the software architecture and their importance to the software's stakeholders. The system requirements with the highest ratings were analysed and based on the analysis the design was constructed so that it ensures that the SA workbench has the following characteristics: support for collaborative work, supports easy addition of plug-in tools, it is portable, and it is constructed of reusable components.

## 7.3 SA Workbench Software Architecture Design

### 7.3.1 Support for collaboration

The architectural design of SA Workbench was based on the design requirements outlined above. The requirement that the workbench should support collaborative work translates into the requirement that the physical machines, where instances of the workbench reside, should be able to communicate with each other. There is a number of architectural styles which satisfy this requirement such as peer-to-peer, master-slave, and client-server architectural styles. The master-slave architectural style suggests that there would be a master machine controlling the communication between the nodes participating in the system. This, however, was not acceptable for the designed system since the system required all nodes to be able to initiate the communication. The client-server architectural style has been identified as more appropriate for the needs of this project than the peer-to-peer architectural style due to its advantages related to ease of maintenance, higher security, and centralised data access. We have selected the 3-Tier client-server architectural style since it overcomes some of the disadvantages of the traditional 2-Tier style related to extensibility, scalability, and reliability [47]. High level logical view, which is used for representation

of the functional requirements of the system and shows what services are provided to the end user, of the initial design is shown in Figure 7.1.

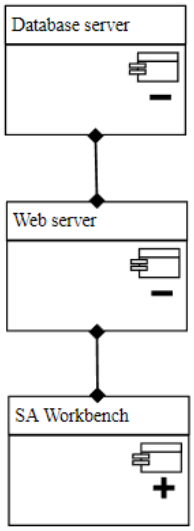


Figure 7.1: High level *Logical view* of the SA Workbench system – SA Workbench Logical view notation.

### 7.3.2 Usability and Portability

The usability and portability requirements influenced our decision to use a web browser on the client side instead of a desktop application. The rendered by the web browsers applications are consistent across different operating systems which is not always the case with the desktop application. Additionally, some programming languages, such as C and C++, do not allow for development of portable application. Java applications are addressing the issue of portability but require installation of the Java Virtual Machine (JVM) which does not satisfy our usability requirement.

### 7.3.3 Extensibility and Reusability

In order to address the requirements for extensibility and reusability we have incorporated the component-based architectural style into our architectural design. The component-based architectural style focuses on decomposition of the design into individual functional or logical components which provide interfaces for communication. This architectural style uses components that are encapsulated, independent and

not context specific which allows them to be reusable, replaceable, and extensible. Additional benefits of the incorporation of this style are that it mitigates technical complexity which leads to decrease of development effort and reduction of costs [47]. The logical view of the system with all of its components is shown in Figure 7.2.

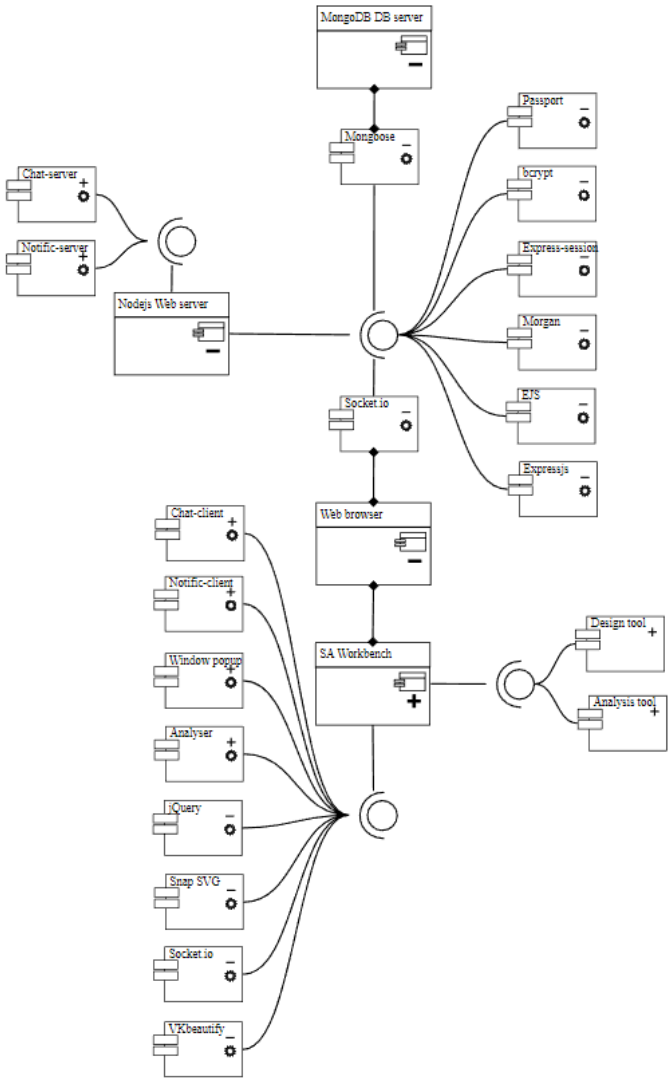


Figure 7.2: Detailed *Logical view* of the SA Workbench system – SA Workbench Logical view notation.

The 3-Tier software architecture comprises of server and client sides. On the server side, we have the web server and database server and on the client side a web browser

and the workbench with its components and tools which are plugged in to it. A Physical view, which is used for representation of the mapping between the software and the hardware used by the system and its distribution, of the architecture is shown in Figure 7.3.

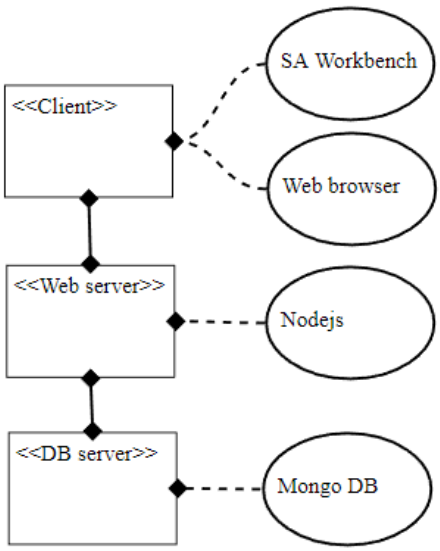


Figure 7.3: *Physical view* of the SA Workbench system – SA Workbench Physical view notation.

The workbench has been designed to have modules residing on the client and server side of the application. When a user performs an action using a tool which has been plugged in to the workbench the tool is communicating with the workbench which in turn communicates with the web browser. An example of this is a user activity related to change of canvas colour. This process is outlined in Figure 7.4 by the use of a process view, which is used for representation of the concurrency and synchronization aspects of the design.

The required functionalities for collaborative work are provided via chat and notification services. Both services have server and client-side parts and share the same architectural design. The architectural design selected is a combination of the publisher-subscriber messaging pattern and the blackboard pattern. In the publisher-subscriber messaging pattern the publishers are objects which use a middleware to fire events after a specific action is performed and the subscribers are the objects

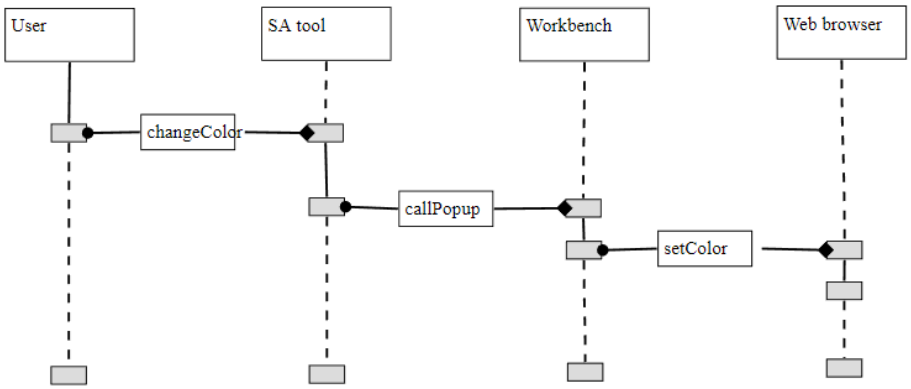


Figure 7.4: *Process view* of user action – SA Workbench Process view notation.

being notified. In the case of the workbench designed the subscribers are users who are part of the users working on the same project as the publisher who fire events every time they make a change to the design. In this pattern, the publishers do not need to know about the subscribers which results in design with loose coupling between the participating objects which improves the flexibility and scalability of the system. Blackboard is a data-centred architectural pattern which is applicable in situations where multiple users work on the same problem. This pattern allows multiple independent programs, also called "knowledge sources", to communicate exclusively through a shared global data repository known as blackboard [48].

When a user makes a change, while using the design tool, to the software architecture design the change is recorded into the shared global database, also known as blackboard, and all of the registered subscribers, the other users registered to the project, are notified. In case that a user works on the same view amended their system requests the amendments and their view is updated. Thus, the amendments are automatically saved and propagated. In case the given user working on the same project is currently working on another view a notification is displayed and no further action is taken. This process is demonstrated by a Process view shown in Figure 7.5.

The open source framework Node.js, which was used to build the designed web server, has a standard development structure to which our design conformed. This decision has been made since such policy allows for easy integration of new modules and collaboration with other developers who have experience with the framework in question. This structure is shown by a Development view, which is used to show the structural organization of the software in the development environment in terms of folders, files, packages, and others, in Figure 7.6.

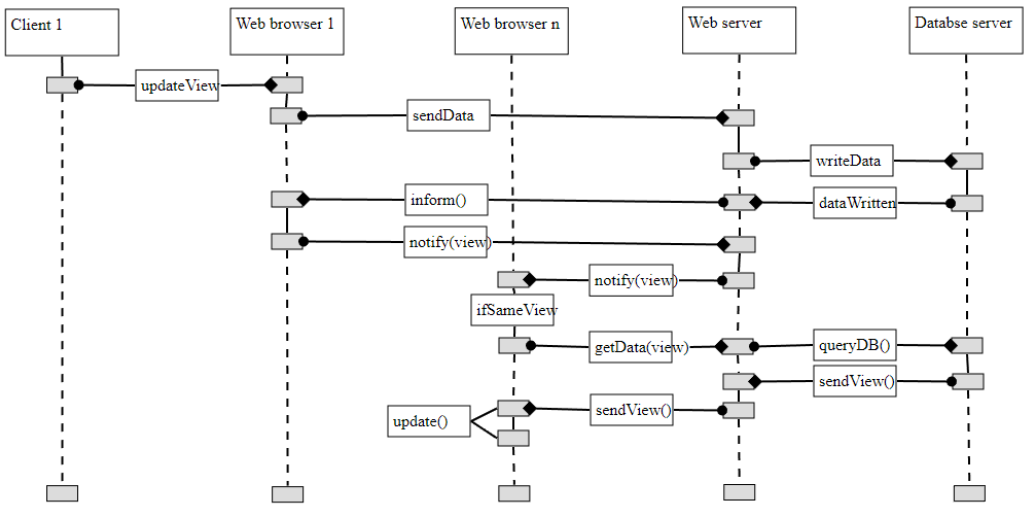


Figure 7.5: *Process view* of user view synchronisation – SA Workbench Process view notation.

The app folder contains database models and the routes.js file which contains instructions related to what action has to be undertaken when a user has sent a request to a given route. The bin folder contains the www script which is the script from which the server is started. The config folder contains configurations for the different parts of the system. The node_modules folder is a standard Node.js folder where the framework installs the specified in package.json file modules. The public folder contains the client-side code of the SA Workbench which is going to be discussed in further detail. Package.json is a JSON file containing meta-data about the web application designed including the modules which are required to be installed. The app.js script is used for importing of the installed modules. The sockets folder contains the server-side code related to the notifications and chat functionality. The views folder contains Embedded JavaScript templates (EJS) files used by the server for rendering of pages associated with them.

The SA Workbench client-side code resides in the public folder. The structure of this folder is presented in figure 7.7. Here the Module folder contains reusable components which are attached to the workbench. The structure of each module is shown in figure 7.8. The SVGs folder contains folders containing the related to the design tool notations in svg file format. Such design has been chosen so that those notations could be used by tools added in the future. The Pages folder contains the pages which are part of the workbench such as the welcome page and the page showing the related to the logged in user projects. The tools folder contains tools which are attached to the workbench.

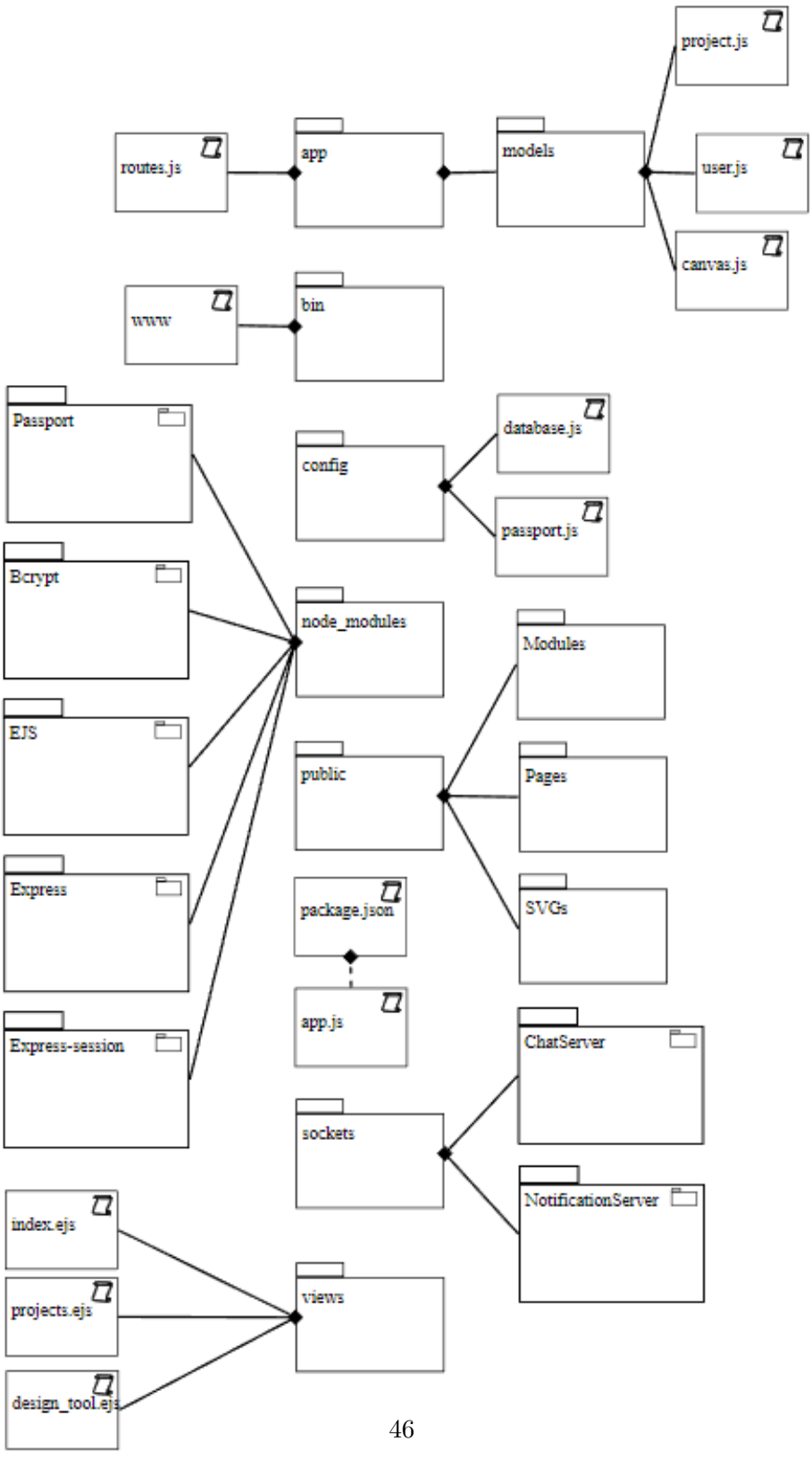




Figure 7.6: *Development view* of server-side code structure – SA Workbench Development view notation.

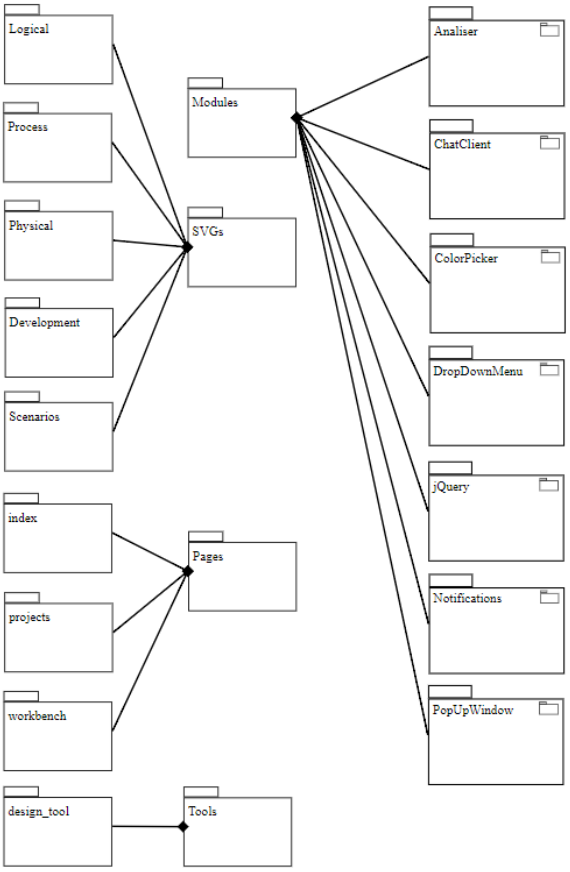


Figure 7.7: *Development view* of client-side code structure of SA Workbench – SA Workbench Development view notation.

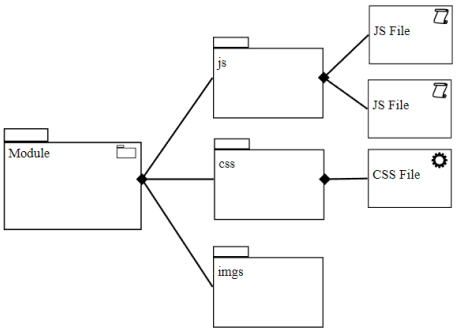


Figure 7.8: *Development view* of client-side code structure of a SA Workbench Module – SA Workbench Development view notation.

The Scenarios are a selection of use cases used for illustration of the decisions made during the development of the other views. Those use cases present the interaction between the designed objects. The Scenarios presented below are documented by the use of the SA Workbench Scenarios view notation which resembles the UML use case notation.

### Scenario 1

A user updates the view they are working on and their collaborators are notified and if they work on the same view their view is updated. When a user updates their view, the system sends the new state of the updated view to the web server which passes it to the database server which than saves it. The database server than informs the web server that the data is saved. If the data is saved without errors the web server sends notifications to all of the users registered to this project. When the collaborators receive the message, they check if they work on the same view. If they are currently not working on the same view they display the message and take no further action. Otherwise, they request the updated data from the web server and update their local version of the view. This scenario is presented in Figure 7.9.

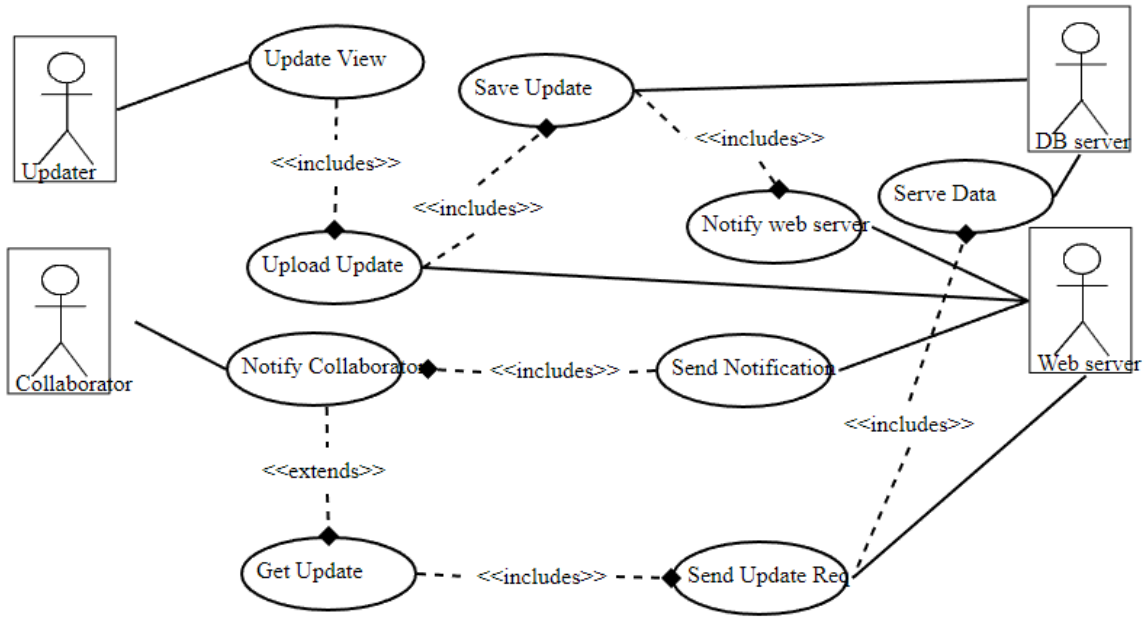


Figure 7.9: *Scenarios view* collaborator notified after update – SA Workbench Development view notation based on UML use case notation .

## 7.4 Conclusion

In this chapter, we showed the documented software architectural design of SA Workbench which is resulting from the ADD's recursive decomposition process which was based on the identified and rated system's requirements. Moreover, we explained the decisions taken in relation to the chosen architectural styles and designed the system's components and sub-components using different architectural patterns and styles such as client-server, component-based, publisher-subscriber, and blackboard. The design decisions taken have been documented using the Kruchten's 4+1 View Model since the proposed views are well understood by the architectural stakeholders, due to its clear structure, and allow us to effectively communicate the software architecture. While the documented design could be further enhanced by additional views and further decomposition of the system's components we believe that overall the resulting architectural design effectively addresses all of the identified system's requirements.