Software Architecture Document (SAD)

Active Requirements Reader

Tom Jochums, Garth Ripton, Courtenay Rojas, Matthew Schwartz

Revision 1.3
March 13, 2013

Revision History

Version	Description of Versions / Major Changes	Responsible Party	Date
1.0	Initial Draft in modified template	Matt, Courtenay, Tom, Garth	2/23/2013
1.1	Presentation Layer. Modular guide descriptions drafted	Matt, Courtenay, Tom, Garth	3/5/2013
1.2	Deployment View drafted, Uses relation added to module guide	Matt	3/7/2013
1.3	Entire document review performed by project team. Multiple modifications made inline as we discussed the document.	Matt, Courtenay, Tom, Garth	3/13/2013

Approval Block

Version	Comments	Responsible Party	Date

Table Of Contents

1 Introduction

The software architecture document (SAD) presents various views of the architecture of the Active Requirements Reader (ARR). Stakeholders interested in the background for these views can find the context for these in the first two sections of the document. The first section provides an overview of document organization, defines who stakeholders are, describes the viewpoints used in viewing the architecture, and provides details on how we will document views. The second section summarizes the context of the ARR application - the problem it addresses and the architectural approach to our solution. The architectural views themselves are shown in the third section. We outline the relationships between the views in the fourth section.

Concepts drawn from the SEI architectural documentation methodologies and the IEEE standard on architectural description (IEEE Std. 1471) form the basis of the SAD [SEI 2013 and IEEE 1471]. We have simplified the format of the IEEE templates based on the scope of the OMSE 555/556 practicum, and therefore some sections are omitted, and others are condensed to only the most important information necessary to describe the architectural views. Descriptions of terminology from these sources are provided in the glossary. For definitions that are more complete and discussions of architectural concepts that are detailed, please see the references.

1.1 Document Management and Configuration

The project team maintains and updates this document collectively. This means that the team is responsible for control of the document. Since the requirements for implementing an “active requirements tool” are broad and not completely understood, we plan to follow an iterative change process that implies the architecture will evolve over iterations. This process is dynamic and we anticipate using and enhancing the SAD concurrently [IEEE 1471]. This makes document configuration control critical. The document will be version controlled with the significant version differences described in revision history above. We will use We will use the Google Docs revision feature to maintain old versions of the document for reference.

1.2 Intended Audience

- OMSE 555/556 Project Team: To control the downstream design efforts by stating the structure of the software elements and how they interact.
- OMSE 555/556 Practicum Advisor: To communicate intentions to external stakeholders and to provide an artifact for review to assure we are making reasonable architecture related decisions.
- Future Students and Project Teams: To evaluate whether the ARR can be used as a baseline for new tools.

1.3 Scope and Purpose

This SAD specifies the software architecture for the Active Requirements Reader (ARR). All information regarding the software architecture may be found in this document, although much

Table of Contents

of the information is incorporated by reference to other documents. The purpose of the ARR project as well as the requirements that form the context in which decisions about architecture are covered in these documents. Examples are the project proposal, software management plan, and software requirements specification in the project repository [RedMine-ARR].

The purpose of the SAD is using our best understanding of the structure of the system to capture and describe the perspectives of the system in a document. Experience has shown creators of software systems that these perspectives are best presented in multiple, complementary views which address a set of concerns.

Bass describes software architecture as the “structure or structures of that system, which comprise software elements, the externally-visible properties of those elements, and the relationships among them” [Bass 2003]. Note that architecture deals with the "externally visible” properties and relationships among software elements. Hence, this SAD focuses on these properties and relationships and leaves the descriptions of internal behavior of the elements to more detailed design documents. The description of the behavior of each element included in the SAD is limited to what other elements need to know to interact with and use it. Each element shown in the architectural views below is an abstraction that omits information that does not concern interactions between it and other elements. These structures will be represented in the views of the software architecture that are provided in Section 3.

1.4 How the SAD Is Organized

This SAD is organized into the following sections:

- **Section 1 (“Introduction”) provides information about this document, its intended audience, and how we will document the ARR architecture.** It provides a guide to the document and overview information like the purpose, owner, and audience. Readers will find descriptions of the project stakeholders and their concerns as they relate to decisions about architecture.

- **Section 2 (“Architecture Background”) explains the context of the ARR needed to understand the decision we have made in creating its architecture.** It gives a general overview of the active requirements review process and how it has steered the direction of the software structure. This includes the constraints and external influences on project.

- **Section 3 (“Views”) and Section 4 (“Relations Among Views”) specify the software architecture**. Views specify elements of software and the relationships between them. The view represents one or more structures present in the software system (see Section 1.3). The document (in Section 4) compares and contrasts the views.

- **Sections 5 (“Referenced Materials”) and 6 (“Directory”) provide reference information for the reader.** The sections provide a list of citations for further description architectural documentation and material needed to find individual elements (an “index”) and definitions of unfamiliar terms or acronyms (“glossary”).

1.5 Architectural Goals

The table below summarizes the architectural goals described in the SRS document and how these goals are addressed in the SAD.

Goal	How this goal is addressed in SAD
Availability	Deployment View addresses how modules interact allowing analysis of potential risks while the software is in operation.
Security	Layered View addresses abstraction to sub-system tiers and their respective logical component. Deployment View addresses how components are hooked together and how they interact and communicate processes or actions.
Usability	Use Case View addresses a user's interaction with the system and depicts the specifications of a use case.
Conceptual Integrity	Layered View addresses abstraction to sub-system tiers and their respective logical components. Module Guide/Uses View addresses conceptual integrity by identifying areas of concern and grouping them into logical presentation, business, data, and service layers as appropriate. Deployment View addresses how the installed application modules will be deployed and communicate with each other at runtime across hardware. Deployment View: (Matt) Addresses how the software components will be deployed across and communicate with each other over the hardware elements. Use Case View addresses consistency and coherence of the overall design.
Maintainability	Module Guide/Uses View addresses how modules fit into the overall system, and what other modules need to integrate with during development. Layered View addresses separating concerns across the different layers while giving the ability to make changes to the layer components isolating any unintended effects in the upper or lower level layers.
Performance	Deployment View addresses how the different processes and hardware communicate with each other.

1.6 View Rationale and Stakeholder Representation

Based on the architectural drivers (goals) identified above the ***Module Guide / Uses***, ***Layered***, ***Deployment***, and ***Use Case*** Views were selected to represent the architecture. The paragraphs below explain the rationale for selecting these views. Each view illustrated in this document also provides detailed background information.

1.6.1 Module Guide/Uses View

Stakeholders: Project Management Team, OMSE 555/556 Project Team, Architecture Team, Quality Assurance Team, Future OMSE 555/556 Teams.

The Module Guide/Uses View is provided to give an understanding of how modules fit into the overall system, and what other modules need to integrate with during development. This guide aids in meeting the conceptual integrity and maintainability goals as stakeholders such as future students and/or project teams and the OMSE Professor may use the guide as a starting point reference for how the system was designed and how any future changes will affect the flow of secrets and services throughout the system.

1.6.2 Layered View

Stakeholders: Project Management Team, OMSE 555/556 Project Team, Architecture Team, Future OMSE 555/556 Teams.

The Layered View will be the primary view as it provides a clear view of abstraction to sub-system tiers and their respective logical components. This view supports particularly the maintainability, security, conceptual integrity business goals by separating concerns across the different layers while giving the ability to make changes to the layer components isolating any unintended effects in the upper or lower level layers.

1.6.3. Deployment View

Stakeholders: Project Management Team, OMSE 555/556 Project Team, OMSE 555/556 Professor, Architecture Team, Implementation Team, Quality Assurance Team, Future OMSE 555/556 Teams, end-users.

The deployment view provides the ability to visualize the how the system should be installed, and how the installed applications will communicate with each other at runtime. This view supports the availability, security, and conceptual integrity business goals by allowing developers to analyze potential risks and while the software is in operation. As a result of this risk analysis, developers can plan for implementing countermeasures for adding robustness to the system.

This view shows interactions as connectors showing how components are hooked together and how they communicate processes or actions. This view is most useful for stakeholders (described above) to determine how modules interact providing an insight into execution and data flow and control.

1.6.4. Use Case View

Stakeholders: OMSE 555/556 Project Team, OMSE 555/556 Professor, Architecture Team, Implementation Team, Quality Assurance Team, Future OMSE 555/556 Teams.

The Use Case View captures system functionality as seen by users. It helps illustrate the use cases and scenarios that encompass architecturally significant behavior, classes, or technical risks. This view supports the usability and conceptual integrity business goals by providing a vehicle used by end users and stakeholders (described above) to discuss the system's functionality and behavior and better determine a timeline for deployed functionality.

1.7 How a View is Documented

All views are documented using the *primary presentation*, *element catalog*, *architectural background*, and *variability guide* description from the Bass and Clements textbook used in the OMSE Architecture Class. Other architecture documentation sections recommended by the book have been omitted due to the limited scope of the OMSE 555/556 practicum.

2 Architecture Background

2.1 Problem Background

2.1.1 System Overview

The Active Requirements Reader Software Requirements Specification provides an overview of the active requirements domain and the what we mean by the active review process.

2.1.2 Goals and Context

The Active Requirements Reader Software Requirements Specification provides the context for and major goals of the project.

2.1.3 Architectural Approaches

The architectural background of the ARR is covered in more detail in the Architectural Background section of the module guide.

3 Views

3.1 Module Guide

3.1.1 Primary Presentation

3.1.1.1 Behavior Hiding Modules

Login View
Service: Collects login input from the user.
Secret:
- How to format markup for the Web Browser to display the login screen.
- How to provide login input from the user to the Web Browser to be authenticated by the API Server.

Account View
Service: Collects and displays account information from the user .
Secret:
- How to format markup for the Web Browser to display the account screen.
- How to provide account input from the user to the Web Browser to be persisted by the API Server.

Review Session List View
Service: Provides a display for the user to navigate through a list of review sessions.
Secret: How format markup for the Web Browser to display the Review Session List screen.

Review Session Editor View
Service: Collects input and displays output for the user to edit a review session document.
Secret:
- How to format markup for the Web Browser to display the Review Session Editor screen.
- How to use the Review Document Editor Script to process requirements text input from the user.
- How to send the processed input back to the Web Browser to be persisted by the API Server.
- How to use the Questionnaire Editor Script to process questions input from the user.
- Hot to send the processed input back to the Web Browser to be persisted by the API Server.

Questionnaire View
Service: Displays a review session with associated questionnaire and collects the questionnaire answers/input from the reviewer.

Secret:
- How to format markup for the Web Browser to display the Review Session Reader screen. How to provide questionnaire input from the user to the Web Browser.
- How to use the Questionnaire Handler Script to process answer input from the user.
- How to send the processed input back to the Web Browser to be persisted by the API Server.

- How to use the Feedback Handler Script to process feedback input from the user.
- Hot to send the processed input back to the Web Browser to be persisted by the API Server.

Forum View
Service: Provides a display for the user of discussion threads and collects discussion input for a review session.
Secret:
- How to format markup for the Web Browser to display the Review Session Discussion screen.
- How to use the Discussion Handler Script to process feedback input from the user.
- How to send the processed input back to the Web Browser to be persisted by the API Server.

Presentation Tier Scripts
Service: Manages changes to the review session document.
Secret: Performs client-side logic in order to parse the users input from the Web Browser into the domain-based requirements content format.

Account Controller
Service: Main entry point of the server used to handle requests to manage account information.
Secret: How to use the services offered by the account manager in order to create and manage an ARR account.

Review Session Controller
Service: Main entry point of the API server which handles requests to create, edit, and participate in review sessions.
Secret:
- How to use the documents generators in creating new review session and questionnaire documents
- How to use the document repository to store new, updated, and feedback related data from the review session.

Questionnaire Controller
Service: Main entry point of the API server which handles requests to answer and give feedback to a review session questionnaire.

Secret:
- How to use the account repository to store new, updated, and feedback related data from the review session questionnaire

Forum Controller
Service: Main entry point of the API server which handles the discussion posts provide prior to the completion of the review session questionnaire

Secret:

- How to use the documents generators in creating new review session and questionnaire documents
- How to use the document repository to store new, updated, and feedback related data from the review session.

Chat Service
Service: Initiates and oversees the connection of two participants in a real time conversation via text “chat”
Secret: How to create a connection between two participants and display and store the conversation thread(s)

Notification Sender
Service: Sends messages to the users of the system.
Secret: How to send out messages through the Notification Server.

Notification Generator
Service: Composes notification messages in response to requests/events
Secret: Knows the format and structure of the notification messages.

3.1.1.2 OS Hiding Modules

Web Browser
Service: Client machine’s software application for retrieving, presenting and traversing Http Requests to and from the API Server.
Secret: How to render web based markup languages and script for the user.

API Server
Service: Hosts the main ARR services.
Secret: How to accept and manage Http Requests from the Web Browser and pass them through to the controllers.

Notification Server
Service: Provides means to transfer notifications between users and between the requirements session services
Secret: Know how to use the system services (HW/SW) to send and receive messages and other notifications

Document Database Server
Service: Provides the service that will define how we connect to the database
Secret: How to storing inform in the database.

3.1.1.3 Software Design Hiding Modules

Account Monitor
Service: Notifies the right account service when state of a user account changes.
Secret:

- How to listen for account changes from the Document Database Server's client API.
- How to wire up specific events to the account services that will act upon them.

Review Session Monitor
Service: Notifies the right review session service when state of a review, question or comment changes.
Secret:

- How to listen for review session document changes from the Document Database Server's client API.
- How to wire up specific events to the review session services that will act upon them.

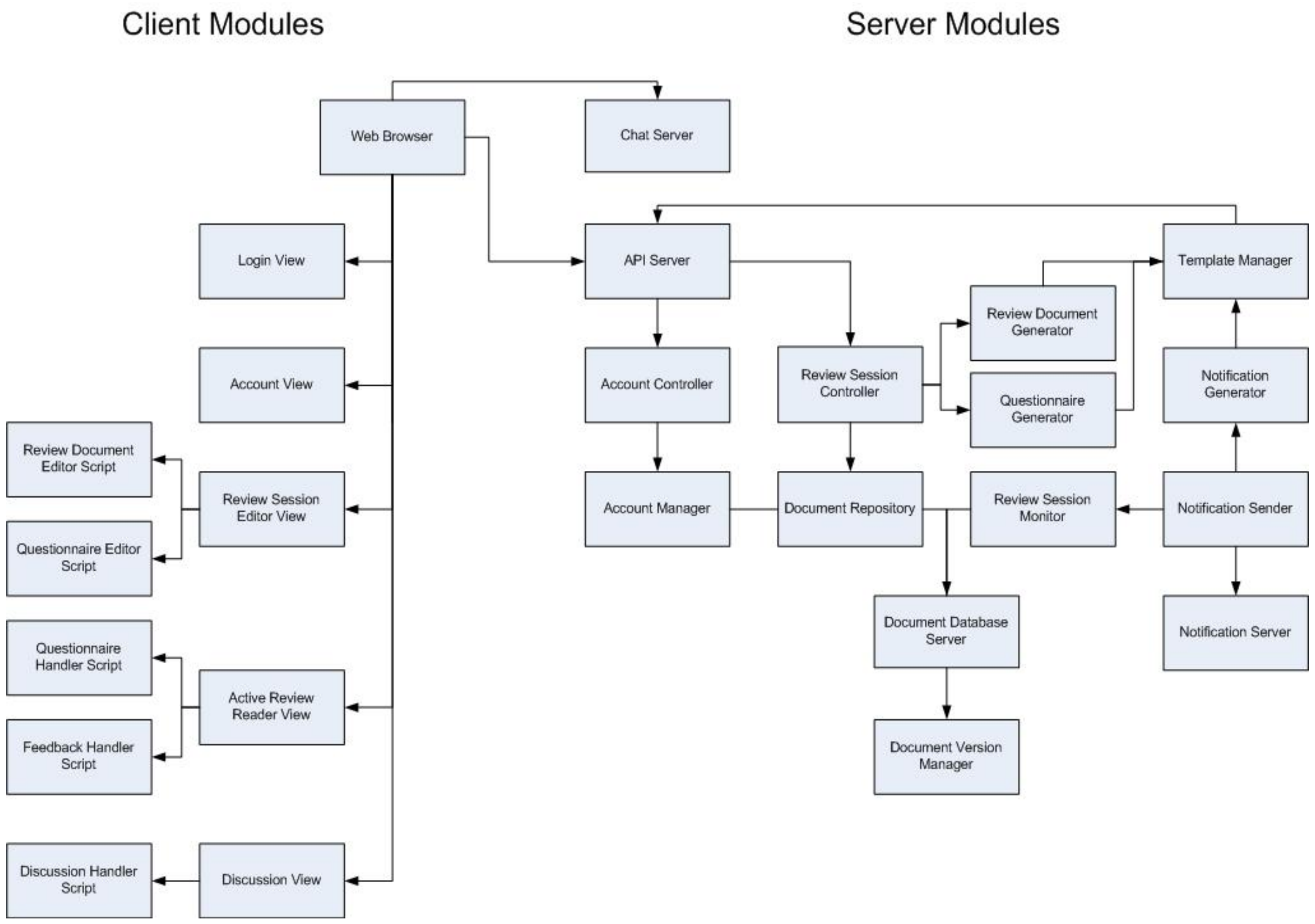
Account Repository
Service: Stores the user account information.
Secret: How to use the Document Database Server's client API to store data.

Review Session Document Repository
Service: Stores the review session document.
Secret: How to use the Document Database Server's client API to store data.

Database Client API
Service: Communicate programmatically with the database.

Document Version Manager
Service: Manages the states and the state changes of the reviews session document (content, questions, answers, feedback)
Secret: Performs document version control logic.

3.1.1.3 Uses Relation Diagram



3.1.2 Architectural Background

The ARR module guide is first and foremost a discovery document for the architects, designers and engineers of the ARR system to figure out how responsibilities of the different modules of the system are allocated. In exploring the architecture, the project team started out defining the different modules of the system and how they interacted with one and other. From that point, we were able to dive deeper into the architectural details of system, deriving the uses, layered, and deployment diagrams that follow. We chose the module guide as the starting point because it has been a key reference document throughout the OMSE curriculum, and we believed that its value as a future guide to design and implementation will make the next phases of the project much easier as they unfold. While there was some difficulty in trying to fit a more modern, web-based system into template that the module guide provides (I.E. Behavior, Device, and Software Hiding elements) , the exercise served its purpose well as many of the system components and their interactions became a lot more clear to the project team after the module guide was complete.

The project team went through many iterations of trying to figure out what all were the modules involved in the system and what secrets they should hide, therefore, trying to explain why all the alternatives were rejected would be somewhat exhaustive. Our major problem areas centered on the modules which were involved with separating the boundaries of the different system processes such as the controllers, persistence related modules, and the specific modules for collecting user input from the user interface. For example, we went back and forth a couple times in deciding whether or not to place the idea of the “view” on the server or client side of the system, a decision which would ultimately affect how the top modules on the *uses* relation diagram were organized. As depicted in the diagram, the secret hiding flows downward from the *Web Browser* and *API Server* modules, pushing decomposition down through to the Database and Notification Servers where the resulting data involved in the system is either persisted and/or sent out to the end user as a message. This pattern of decomposing secrets through from the presentation, to the messaging, to the data tier of a system is fairly well known and used through the internet software industry.

Moving forward , the guide will be used by the project team in the future to break up future development of the system into units of work. The ARR system designer / engineer will find the module in the guide that they are responsible for, and make note of what secrets they must hide in design and implementation. They will use the *Uses Relation Guide* to understand how their modules fit into the overall system, and what other modules they will need to integrate with during development. Finally, the guide may be used by future students and/or project teams as a starting point reference for how the system was designed and how any future changes will the flow of secrets and services throughout the system.

3.1.3 Variabilities

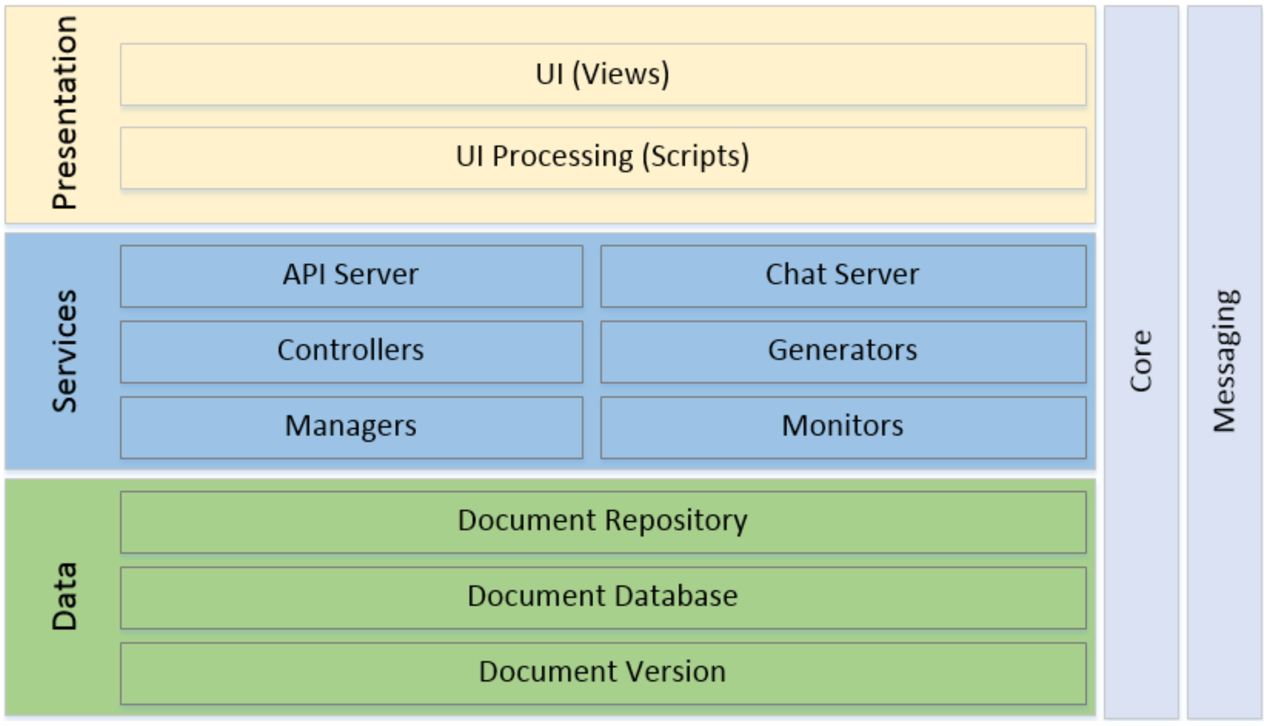
As the module guide has been the driving model for the rest of the architectural views of the system, the project team spent enough time so that there are few if any unbound decisions left to consider. Any remaining variabilities would most likely have to do with the secrets that the modules hide rather than any omitted modules or how the existing modules are structured and interact with each other. For example; the Document Database Server has plenty of secrets that are not listed in the module guide, some of which we may have to document in the future if we plan to implement the Database Server ourselves. However, the plan for now is to use a third party software library for the database and therefore the variabilities for secrets for that particular module are somewhat unknown at the present time. A short of example for more of these possibilities are as follows:

- 3.1.3.1: Database Server Secrets such as how the data is stored in memory.
- 3.1.3.2: API Server secrets such as how http Request are logged, how errors are

handled, how internet connectivity is configured.
- 3.1.3.3 Web Broswer secrets and services such as what versions of HTML, Javascript, and CSS are supported.

3.2 Layered View

3.2.1 Primary Presentation



3.2.2 View Elements

Elements	• **UI**: Visual elements of the application that the user will see when they are interacting with the software. This could be any visual framework whether it be HTML, or some other. In our module guide this is the set of the views that we've outlined. • **UI Processing:** Provides the communication back to the Service tier to actually process the user input so that the application can act on that input. Ultimately will be requesting information and passing information into the Services Layer • **API Server:** Provides one point of external communication for the Presentation Tier to connect to in order to retrieve information for the user and process information from the user.

	• **Chat Server:** Provides a service to the Presentation layer to allow for chat communication between participants who are interacting with the presentation tier at the same time. • **Controllers:** Handles the interactions as they come in from the servers in order to pass control through to the necessary generator and manager modules within the system. • **Generators:** For the more complex objects in the system that will need to be created and have larger sets of objects that get created. • **Managers:** For the more complex objects in the system that might have a succession of or collection of objects that will be getting interacted with, these managers will help with this interaction. • **Monitors:**Special kind of controllers that listen for changes to the underlying model so that they can relay information throughout the rest of the system.Helps deal with the real time notifications that would happen from other users of the system. • **Document Repository**: Maintains the basic data model underlying the application and handles the coordination with the database to persist the model. Deals with any conversion of the Domain Model that the rest of the application interacts with to the database model for persistence. • **Document Versioning Management:** Provides the versioning control of documents as they are stored within the system. • **Document Database:** Actual store of the data that is represented by the data model. Is considered separate from the domain model because the database won't necessarily map perfectly onto the model that the rest of the application interacts with.
Boundaries	• **Data Tier**: The tier that handles the underlying data that is being acted upon by the application, and the model that is associated with it. We've got two concepts down at the model level which are the entities associated with the application domain, and the actual store of the data that the user will be using. • **Presentation Tier:** The presentation of the data to the user, this is the front end that the user will interact with. It is generally light on logic pertaining to the application underneath, and focuses on the logic of how to display the domain model to the user, how to get information back from the user. Includes some of the processing functions that will be necessary for the user interaction with the rest of the system. • **Services Tier:** Handles the interactions that will happen between the presentation tier and the data tier. Will provide a lot of the creation and modification of the domain model that is being presented, will also include any validation that wouldn't normally happen at the presentation layer. • **Core:** Vertical slice that deals with the basic management of the system covers things such as logging, and the basic security that we will be using

	throughout the system. ● **Messaging:** Vertical slice that covers the communication services that are actually being used between the presentation, services, and data tiers. It signifies more the technologies that would be used to handle that communication.
Relationships	● **Presentation Tier - Services Tier:** The interactions between the presentation and the services are the presentation layer telling the services what information it is wanting to display, and what interactions the user has taken upon that information. There is also the concept of the services notifying the presentation layer of underlying changes that have been made. ● **Services Tier - Data Tier:** Based on the user's interactions that have been related back to the services tier, the services communicate with the model to change that underlying data at the persistence level, and to retrieve data from the model to pass back up to the view to display to the user. ● **Core - Application:** Application uses the core slice as utilities to handle the shared functionality that all layers of the application will need to use, such as logging and security mechanisms. ● **Messaging - Application:** Application uses the messaging slice as the means for communicating between the layers. The communication layer is essentially masking what the typical communication mechanisms will be between the different tiers if necessary.

3.2.3 Architectural Background

The goal of this view is to provide the logical tiers of the application and how we intend to break down where the responsibilities reside for the application. Initially we'd discussed a Model-View-Controller style architecture, but decided that based on some of the decisions around the technology we were researching, that though similar, doing a tiered architecture would likely result in a better separation of the responsibilities with how we wanted to separate the control of the system. We also have separated any communication between the data layer and the presentation layer so that it has to pass through our services layer, which was another reason to approach this as a tiered architecture.

In addition to the layers that are included in the system, we have some cross-cutting concerns that are addressed by the vertical bars in the diagram. There are a number of utility style functions that we'll want the system as a whole to be able to deal with, most notably, the communication, logging, some aspects of the error handling, and security. As such, we see these needing some standardization across the application so that all parts of the system can deal with these needs without necessarily needing a specific implementation of the other tiers.

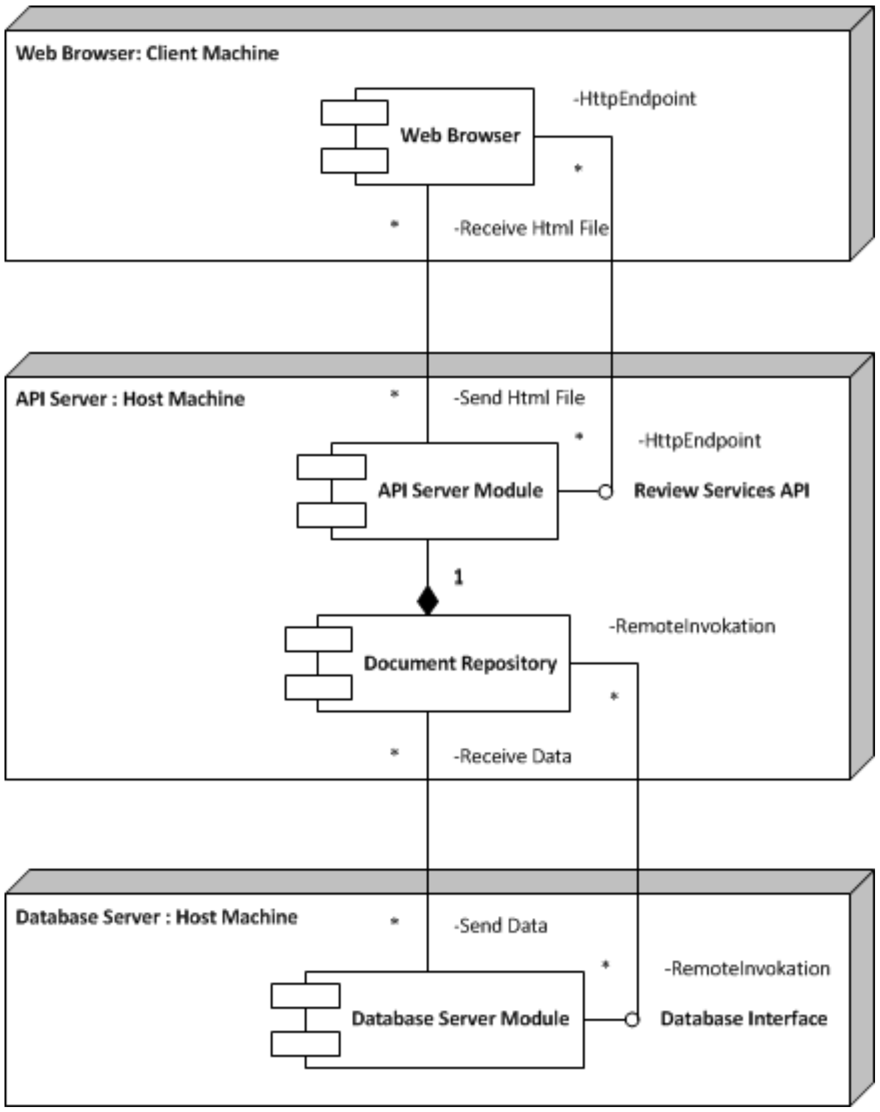
3.2.4 Variabilities

Though most won't likely be dealt with this in the scope of this project, there are a number of variabilities that would be supported with this view depending on whether or not the implementation is done in a proper way. The main variabilities are:

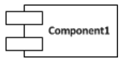
- 3.2.4.1 The user interface: Based on having the view separated out into its presentation layer, depending on whether we want to support a different UX for the view, that could be swapped out and changed
- 3.2.4.2 The data layer: Because we have the model encapsulated, we could potentially change the underlying technology we used under the model to something different if needed at some point.
- 3.2.4.3 Utility mechanisms: Logging specifically if we need to change where and how and what happens when we log messages in the system, this is going to be isolated in the Core slice.

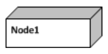
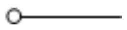
3.3 Deployment View

3.3.1 Primary Presentation



3.3.2 View Elements

Elements 	● **Web Browser**: The software application installed on the Client Machine used for retrieving, presenting and traversing HTTP information resources across the internet. The browser serves as the ARR system's client, rendering the display of the user interface which is sent to it via the ARR System as a collection of HTMl files. The browser will also execute the calls to the ARR API

	Server using Javascript, as initiated by the end user. ● **API Server:** The software application installed on the Host Machine used for exposing the main interfaces for the backend ARR system. This program will contain the business logic for processing review session information and communicating with the client's web browser. The API server will also read and write data via the Document Repository. ● **Document Repository:** As a component of the API Server, the document repository will provide an abstraction layer over communicating with the database. This software library will be responsible for generating the commands necessary to connect, query, and read and write data from the Database Server. ● **Database Server**: The software application installed on the Host Machine (could potentially be installed on another machine - as long as there is an open channel of communication between it, and the host machine.) which is responsible for persisting and versioning the ARR review session data.
Boundaries  	● **Client Machine**: The physical hardware on which the Web Browser is installed. This could be any type of hardware on which normally a web browser is installed and operated. ● **Host Machine:** The physical hardware on which the API Server and Database Server are installed. The hardware and operating system configuration will be determined by the API and Database Server software.
Relationships 	● **Web Browser - API Server:** The Web Browser and API Server will communicate with each other over HTTP, providing the main entry point of the system. The API Server will be configured to allow Http Requests to be made to it by the Web Browser under any of the following mechanisms: 1. As published to the world wide web and made publicly available. 2. As published to the world wide web and secured by IP. 3. As installed as an internet application. ● **API Server - Document Repository - Database Server:** The API Server will communicate with the Database over a remote protocol as defined by the database interface. It is assumes that the database developer will provide a software library (installed with the API Server) which abstracts this remote communication, which will be and called by the API Server's Document Repository module.
Interfaces 	**(High level description only required for deployment, low level to be documented in design)** ● **Review Services Interface:** The review service interface will be exposed as an HTTP endpoint accepting and sending textual data as it input and output. It's data type definitions will be described in a textual contract (most likely using XML, JSon or some other widely accepted format) and will be constructed by the API Server Software and the Web Browser. ● **Database Interface:** The database interface will be defined by the database manufacturer and will be exposed to the API Server software library over a

	custom written 3rd party software library that is installed on the database server. To the this diagram is concerned with the database interface, it is assumed that object data will be serialized into binary or textual data by the software installed on the API Server, and then sent to the interface over a remote protocol.

3.3.3 Architectural Background

The primary focus of this view is to describe how the distributed nature of the system is achieved. The web and internet are distributed environments in nature, and the deployment diagram describes how the different components of the system communicate with one another over the internet. An alternative approach could have been to use more of a low level, peer to peer connection where the data was passed from client machine to client machine, but that would have limited the availability of the system which was one of the founding goals of the system. We also recognized that the skill set of the engineering team leaned heavily towards internet protocols and technologies, so some of the decision to involved web development probably was influenced by this as well.

The *nodes* in the diagram are layered by separation of physical hardware. Installing the system will involve installation of both client and server software, so we thought that the deployment diagram could double as a client-server diagram as well. We assumed that the database layer would be installed on the same machine, and with the same installer as the API software, but we wanted to leave open the flexibility to possibly have the database installed separately, on a different machine. This would provide us with the potential to become scalable if we decided to change the design to cater to a large number of organizations in the future.

The last design element that we thought to include in the deployment document was the *Document Repository.* While this component would most likely not be installed separately from the API Server, it is important to describe how to communication with the database server is abstracted from the API Server because if the database interface ever changed then the resulting design may call for a Data Service API that could potentially be installed on the database server itself.

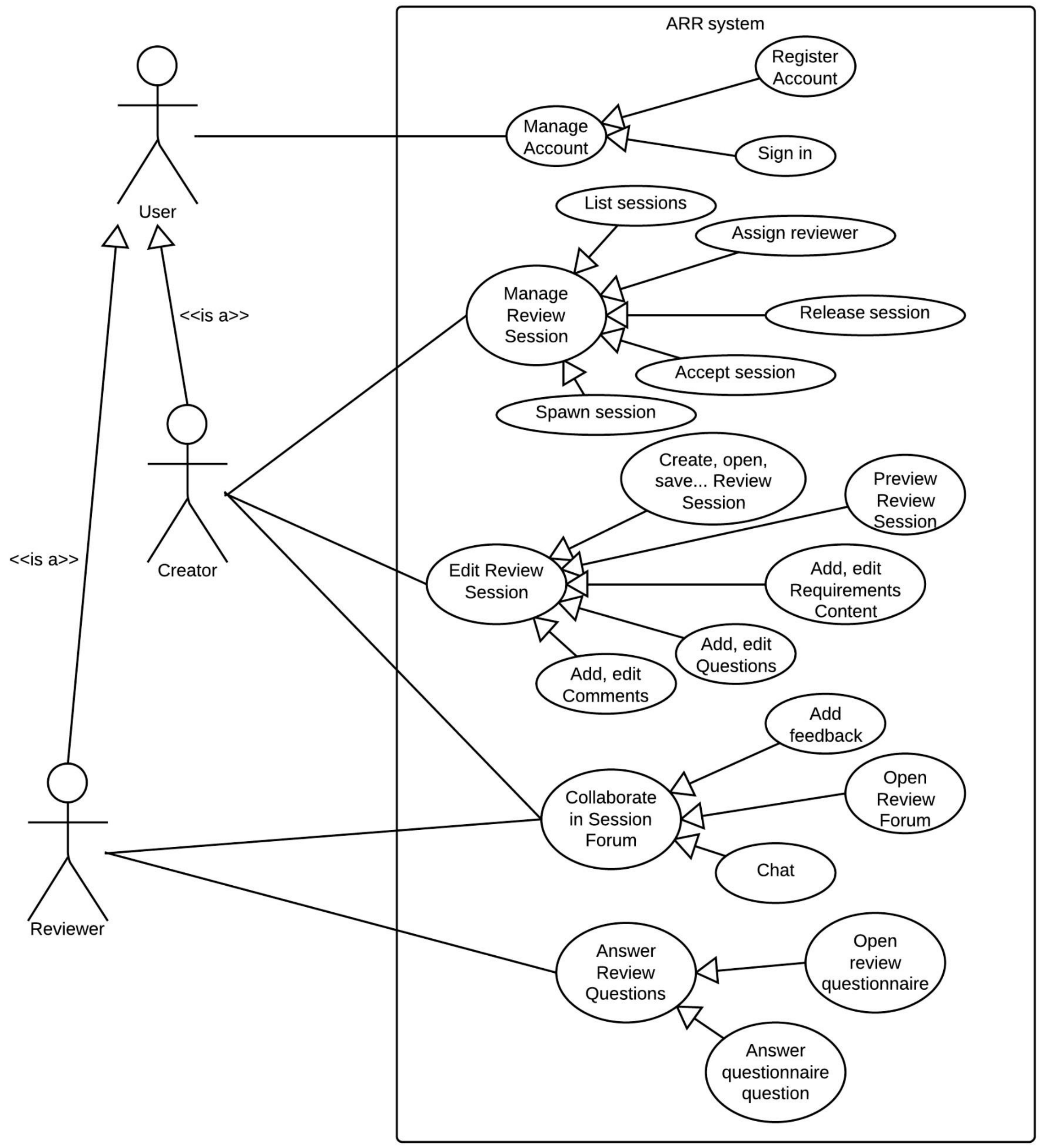
3.3.4 Variabilities

Variabilities are few with the limited time of the OMSE 555/556 practicum, but potential points are as follows:

- 3.3.3.1: The support of multiple types of web browsers
- 3.3.3.2: The fomat of data passed over HTTP between client and sever.
- 3.3.3.3 Embeded versions of the database server are available we it may live in the same process as the API software.
- 3.3.3.4 Multiple databases may be employed as the document repository database that we are looking at inititally does not support a relation heirarchy.

3.4 Use Case View

3.4.1 Primary Presentation



3.4.2 View Elements

Elements Actors	● **User**: any person that interacts with the ARR. In this case, either the creator of an active review session or a reviewer of an active review session is a user. This actor needs the services required to manage the user access and settings. ● **Creator**: a person that needs to create, populate, and release active review session that can be examined by a reviewer. (Note: A creator is a user and inherits the use cases/services of the user actor.) ● **Reviewer**: a person assigned to examine/review the requirements material assembled by the creator of the review. (Note: A reviewer is a user and inherits the use cases/services of the user actor.)
Use cases	(**High level use case**) (*Component use cases*) ● **Manage Account**: services needed to maintain an account on the ARR system and to identify and manager users ○ *Register account*: create and account for a new user ○ *Sign in*: grant access to an existing user ● **Manage Review Session**: services needed to keep track of review sessions as well as send sessions through the specific ARR workflow. ○ *List sessions*: Show all of the review sessions that a creator has the proper rights to access ○ *Assign reviewer*: Determine who will be tasked with a specific review ○ Release session: After requirements material has been added and reviewers are assign, allow reviewers and others to access the material under review. ○ *Accept session*: After review questions are completed and any feedback has been received close out the session ○ *Spawn session*: recreate a prior review session with selected content (selectively duplicate a session) ● **Edit Review Session**: services needed to add content to, change or supplement the material in the active review session as desired by the review creator. ○ *Create, open, save...review session* (three use cases): perform one of the three actions on a review session that the creator can access and wants to edit or has edited. ○ *Add, edit... review session* (two use cases): include new or change existing requirements documentation or other material that the creator want the reviewer to examine. ○ *Preview review session*: allow the creator to check a review session he/she has created and has requirements content, questions or comments. ○ *Add, edit...questions* (two use cases): include new or change existing active questions that the creator want the reviewer to answer. ○ *Add, edit...questions* (two use cases): include new or change existing supplementary comments that the creator want the reviewer to see as part of the review process. ● **Collaborate in Session Forum**: services that connect or allow direct interaction between the ARR creator and the reviewer. ○ *Add feedback*: allow either actor to insert a comment related to the review to a section of the review ○ *Open review forum*: initiate a off line threaded discussion between the creator and review dealing with a specific topic

	○ *Chat*: initiate a real time text conversation between the creator and the reviewer if possible ● **Answer Review Question**: services that allow the ARR review to respond to the reviewer's questions. ○ *Open a review questionnaire*: retrieve and display the set of review questions created by the reviewer for this review session. ○ *Answer questionnaire questions*: allow the reviewer to respond in writing to the reviewers specific question.
Boundaries	● **AFF system boundary**: the "ARR system" box that encloses the use cases forms the border between the system internals and the external user. All of the functionality (services, features) within the boundary are implemented by the system. Other functionality or roles (like actors) are external to the system and are not specified in the ARR design documentation.
Relationships _____ —▷ <<is a>>	● **Actor to high level use case** (solid lines no arrows)- show the services required by the actors with minimal detail. This is the pure "use case relationship"- the actor uses the services of the use case to fulfill the "user" needs that the software is design to implement. ● **High level use case to lower level use case** (lines with open arrow)- these denote use case "uses" relationship between the high level use case that "uses" the lower level use case to implement it functionality. The high level use case is composed of the lower level use cases.)We have omitted the normal label for this relationship "<<uses>>" to reduce clutter on the graph) ● Actor inheritance (lines with open arrow and <<is a>> label)- show that the two actors (creator, reviewer) are "users". As described above the creator and reviewer inherit the use cases/service of the "user" use case.

3.4.3 Architectural Background

The use case architectural view shows the functions and features of the active requirements reader in graphical form. The use cases shown here are directly related to those in the software requirements specification (SRS-in GoogleDrive). The SRS, which defines the behavioral and quality requirements, includes a more detailed description of each use case. Here, we present these use cases in a format that shows the relationships between them. The view also indicates which people (actors) use which element of the functionality.

3.4.4 Variability Guide

There are several optional features/use cases that we may consider but are not include in the primary view:

- 3.4.3.1- Print review session summary- (Manager Review Sessions)- allows the creator to retail a copy of the review in summary form. Could be an electronic(PDF or other) or hard copy.
- 3.4.3.2- Invite reviewer to review session- (Manage Review Sessions)- this is a variation on the "assign reviewer" use case that includes extra functionality to allow a new user/reviewer to register for an ARR system account as part of the assignment process.

There is also the possibility of an additional role/actor that would oversee the “Manage Account” high level use case- the “administrator”. We have chosen to make the initial version of the ARR self-administered. We would anticipate the needs of security and scalability (to multiple teams and multiple organizations) could be met by design centralized control of account creation and access.

3.4.5 Other Information / Glossary

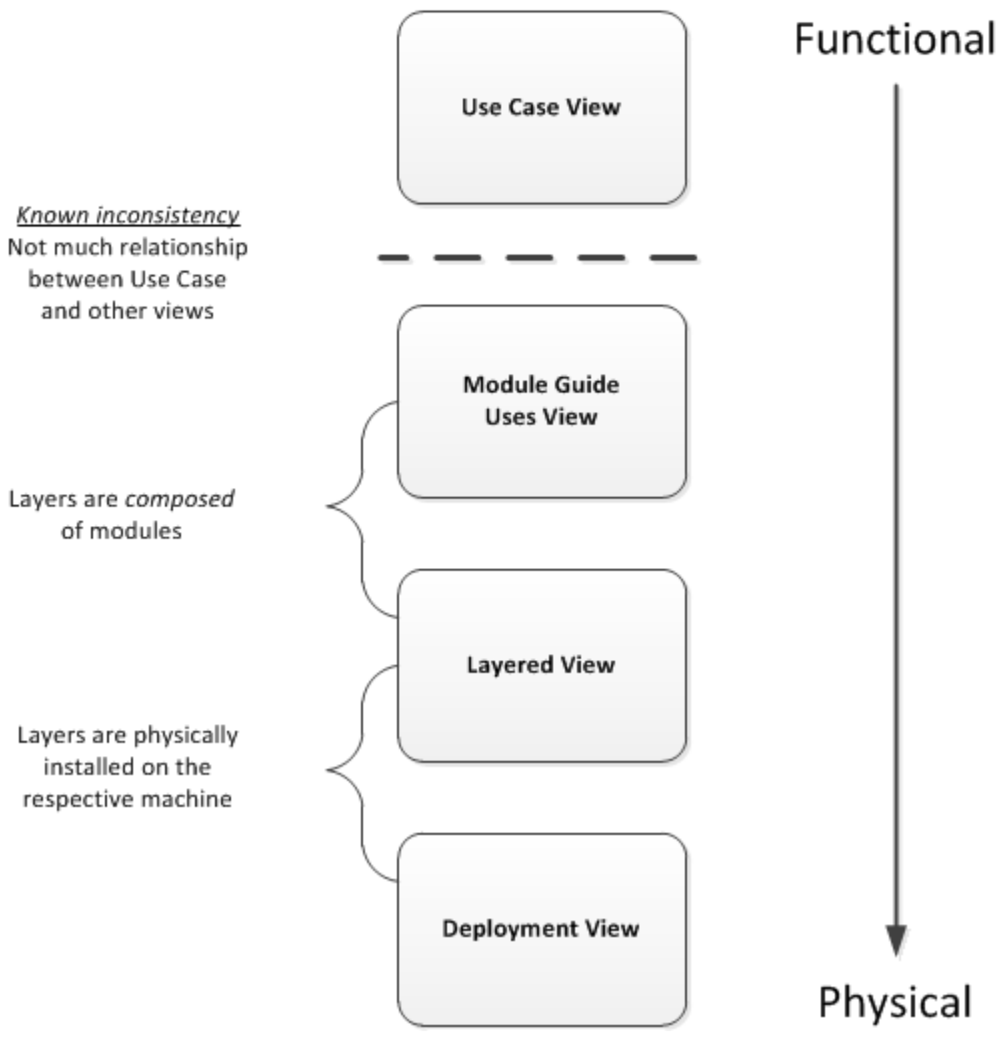
Note that in several cases multiple use cases are combine into a single one. For example, “Create, open, save...review session” is composed of three distinct use cases for each of those functions (the process of creating a new session, opening an existing session, save a modified version of the session).

4 Relations Among Views

4.1 General Relations Among Views

The project team initially discussed using the 4 + 1 architectural view model to drive the relationship between our views but by the time we had finished the first draft of the SAD it turned out that we ended up following a different approach. The first divergence took place right away as we decided to create the module guide / uses view for the document. As stated in prior sections of this document, we did this because we wanted to exercise our knowledge of the module guide in this document as it was such an important part of the OMSE curriculum. If anywhere, the module guide might fit into the “logical” corner of the 4 + 1 view but it’s really isn’t mentioned in Krutchen article.

The relationship between the views then became somewhat informal and organic as we took the information gained from creating the module guide / uses document and decided from there which views to create next based on that information and what architectural goals we wanted best represent. As it turned out, the order of view creation and relationship flowed in somewhat a sequential fashion, starting with the most *functionally* focused view (Use Case) and ending up with the most *physically* focused view (Deployment). The most apparent relationship between the elements of the views are that the elements in a *proceeding* view are generally composed of the elements in the *preceding* view. The following illustration describes this relationship at a high level;


Functional
Use Case View
Known inconsistency
Not much relationship between Use Case and other views
Module Guide Uses View
Layers are composed of modules
Layered View
Layers are physically installed on the respective machine
Deployment View
Physical

As depicted, the Use Case view sits on top of the chart and acts as the most "functional" description of the architecture. While it does not share direct relationships with any other views, the project team felt that the product in general was a very "function/feature rich" software system and we felt that we wanted a good visual perspective of that aspect. The Layered View, which sits atop the Module Guide / Uses View has it's layers *composed* of the modules in the guide. For example; the *Services* Tier is composed of the *Controllers* and *Generators* from the module guide. Finally, the Deployment view describes how the software should be installed on the machines that it will run on. This view sits below the Layered view as *tiers* will be the units installed as packages or executables on the machine (I.E. The *Services* Tier is installed on the Host Server.)

More details about how the relationships that elements share between the views are spread throughout the Element Descriptions and Architectural Background sections of the views themselves. Some elements between the views share names (I.E. Server, Controller, Document Repository, etc..) but should obviously be examined in context of the view where they are described. Pointing out the commonalities abou these elements that are shared amongst multiple views would be an extensive activity and will only be performed if required in a second iteration of the architecture phase of the project.

5 Referenced Materials

Bass 2003	Bass, Clements, Kazman, *Software Architecture in Practice,* second edition, Addison Wesley Longman, 2003.
IEEE 1471	ANSI/IEEE-1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 21 September 2000.
RedMine-ARR	Active Requirements Reader Project Repository located at: https://projects.cecs.pdx.edu/projects/wi2103omse555-arr (Accessed 2/22/2013)
SEI 2013	Software Engineering Institute- Carnegie Mellon University, *Software Architecture, Tools and Methods for Documenting the Architecture* at http://www.sei.cmu.edu/architecture/tools/document/. (Accessed and downloaded 2/22/2013)
Krutchen 1995	Krutchen, *Architectural Blueprints—The "4+1" View Model of Software* Architecture, IEEE Software 12 (6) November 1995, pp. 42-50

6 Directory

6.1 Glossary

Term	**Definition**
View	A representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. A representation of a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.
Stakeholder	An individual, team, organization, or classes thereof, having an interest in the realization of the system.
Active Review	A review in which the reviewer is actively engaged in the review

	via mechanisms to focus their attention. In this application, the focus will be gained by asking specific questions to engage the reviewer.
Creator	A role that a system user can take in order to create a review for to be performed by a reviewer. Responsible for releasing a Review session.
Reviewer	A role that a system user can take when interacting with the system. Will be the one that answers questions during a review session.
Session	The instance of the review that is assigned to a reviewer to be taken. Has a one to one relationship between creator and reviewer.

6.2 Acronym List

Term	**Definition**
ARR	Active Requirements Reader
API	Application Programming Interface
IEEE	Institute of Electrical and Electronics Engineers
OS	Operating System
RUP	Rational Unified Process
SAD	Software Architecture Document
SEI	Software Engineering Institute
SRS	Software Requirements Specification
OMSE	Oregon Masters of Software Engineering
UI	User Interface