Design Document

CEN4010 – Software Engineering
Prepared for Peter Clarke
By Team 5

Teriq Douglas
Yovanni Jones
M. Kian Maroofi
Armando J. Ochoa
Anthony Sanchez-Ayra

November 12, 2019

Abstract

Student Organization System (SOS) is a web-based system meant to provide leaders and administrators of organizations a way to manage members and events. Simultaneously, it allows users to monitor the events and organizations they belong to. Although it is like other organization systems which are hosted and managed by university and colleges, the SOS differs in that it is primarily online and has a primary focus on event hosting and user interaction. The SOS is developed using the Unified Software Development Process (USDP), which is described in the two first sections of this document. The specifications of the system are captured in the form of use cases, which describe the use case model of the USDP and creating UML class and sequence diagrams, which are part of the Analysis model of the USDP. As part of the Design model, this document contains detailed descriptions of the chosen architectures for the SOS (Three-Tier (3TA) and Repository) as well as detailed descriptions of the decomposition of the SOS along with detailed diagrams explaining the functionality of each subsystem. The SOS also uses multiple design patterns (namely, Singleton, Command, Façade, and Builder) to ensure the efficiency and reuse of code in the system.

Table of Contents

Abstract 2
1 Introduction 6
1.1 Purpose of the System 6
1.2 Requirements 6
1.2.1 Functional Requirements 6
1.2.2 Non-Functional Requirements 7
1.2.2.1 Usability 7
1.2.2.2 Reliability 7
1.2.2.3 Performance 7
1.2.2.4 Supportability 7
1.2.2.5 Implementation 7
1.3 Development Methodology 8
1.4 Definitions, Acronyms, and Abbreviations 9
1.5 Overview of the Document 9
2 Proposed Software Architecture 10
2.1 Overview 10
2.2 Subsystem Decomposition. 11
2.3 Hardware and Software Mapping 12
2.4 Persistent Data Management 12
2.5 Security Management 16
2.5.1 Password Management. 16
2.5.2 Access Management 17
3 Detailed Design 18
3.1 Overview 18
3.1.1 SOS Website 18
3.1.2 SOS Interface 19
3.1.3 User Management 20
3.1.4 Event Management 20
3.1.5 Organization Management 21
3.1.6 Security Management 22

3.1.7 SOS Storage 23
3.1.8 Google Maps GPS API 24
3.2 State Machine 25
3.3 Object Interaction 25
3.3.1 Sequence Diagram for SOS01 – Create an Event 26
3.3.2 Sequence Diagram for SOS04 – Attending an Event. 27
3.3.3 Sequence Diagram for SOS02 – Grant Organizer Role 28
3.3.4 Sequence Diagram for SOS07 – Edit Profile 29
3.3.5 Sequence Diagram for SOS16 – Create Organization 30
3.3.6 Sequence Diagram for SOS17 – Cancel an Event 31
3.3.7 Sequence Diagram for SOS22 – Registration 32
3.3.8 Sequence Diagram for SOS10 – Access an Event by Location 33
3.3.9 Sequence Diagram for SOS31 – Login 34
3.3.10 Sequence Diagram for SOS32 – Logout 35
3.4 Detailed Class Design 36
3.4.1 Class Description 36
3.4.1.1 SOS Website 36
3.4.1.2 SOS Interface 37
3.4.1.3 User Management 38
3.4.1.4 Event Management 39
3.4.1.5 Organization Management 39
3.4.1.6 Security Management 40
3.4.1.7 SOS Storage 40
3.4.2 Control Objects Description 41
3.4.2.1 SOS Server OCL 41
3.4.2.2 User Manager OCL 42
3.4.2.3 Organization Manager OCL 42
3.4.2.4 Event Manager OCL 43
3.4.2.5 PasswordManager OCL 43
3.4.2.6 AccessManager OCL 43
4 Glossary 44
5 Approval Page: 45

6 References 46
7 Appendices 47
7.1 Appendix A – Use Case Diagram 47
7.2 Appendix B – Implemented Use Cases 48
7.2.1 Create Event 48
7.2.2 Grant Organizer Role 50
7.2.3 Attending an Event 52
7.2.4 Edit Profile 54
7.2.5 Access Events by Location 56
7.2.6 Create Organization 58
7.2.7 Cancel an Event 60
7.2.8 Registration 62
7.2.9 Log in 64
7.2.10 Log Out 66
7.3 Appendix C – Detailed Subsystem Class Diagrams 68
7.3.1 SOS Website 68
7.3.2 SOS Interface 69
7.3.3 User Management 69
7.3.4 Event Management 70
7.3.5 Organization Management 70
7.3.6 Security Management 71
7.3.7 SOS Storage 71
7.4 Appendix D - Class Interfaces 72
7.5 Appendix E – Diary of Meetings 101
7.5.1 October 7, 2019 101
7.5.2 October 14, 2019 103
7.5.3 October 21, 2019 104
7.5.4 October 28, 2019 105
7.5.5 November 4, 2019 106
7.5.6 November 8, 2019 107

1 Introduction

The following chapter introduces the Design Document (DD) with the goal of explaining how certain qualities of the Student Organization System (SOS) project should be optimized.

The purpose of the Design Document (DD) is to define the decomposition of the system and object design of the SOS to create both a stable and efficient architecture. When decomposing the system, the team first applied two architectural patterns that were found fit for the system. Afterwards, the team decided to investigate object design patterns to ensure that the software of our system would be reusable and easily modifiable. In addition, in this DD we also listed major policy decisions such as control flow, access control and data storage of the SOS.

The purpose of the SOS is defined below. Following that, the scope of the system is defined in Section 1.2. Section 1.3 contains a list of relevant terms, acronyms, definitions and abbreviations used throughout the system. Finally, Section 1.4 contains a brief outline of this document. Following chapters include a detailed section on proposed software architecture (Section 2) and a detailed section on the design of SOS (Section 3).

1.1 Purpose of the System

The Student Organization System (SOS) is a web-based system meant to provide leaders and administrators of organizations a fast, interactive, and accessible way to manage members and events from a single, centralized place. Simultaneously, the SOS system also allow users to monitor and keep up-to-date information about the events and requirements of the organizations they belong to. Finally, the system also allows organizers to advertising their organizations and recruit new members from the general userbase. In essence, the Student Organization System is meant to aid the interaction between members and organizations.

Although the system is meant primarily for academic settings, with Universities being the main target, organization creation and management is open and could be used in other environments, both academic (High Schools, etc.) and non-academic (Company Campuses, Community Centers, etc.).

1.2 Requirements

This section defines the functional and non-functional requirements of the SOS system. A more complete and detailed description of the system requirements can be found in the Software Requirements Document (SRD) for this project.

1.2.1 Functional Requirements

Below is a short description of the functional requirements of the SOS system for each of the implemented Use Cases. The complete use cases for each can be found in Appendix B.

- The system shall allow an organizer to create events for their organizations (see Use Case SOS01 in Appendix B).
- The system shall allow the current organizer to add/invite other members of the organization to be granted with the organizer role (see Use Case SOS02 in Appendix B).

- The system shall allow users to check-in for each event on the platform (see Use Case SOS04 in Appendix B).
- The system shall allow users to edit their profile data including their email, phone number, date of birth, password, and privacy features (see Use Case SOS07 in Appendix B).
- The system shall allow users to find all nearby events based on the user's current location (see Use Case SOS10 in Appendix B).
- The system shall allow users to create their own organization (see Use Case SOS16 in Appendix B).
- The system shall allow the organizer to cancel the event (see Use Case SOS17 in Appendix B).
- The system shall allow visitors to register for a new account (see Use Case SOS22 in Appendix B).
- The system shall allow users to login to their registered account (see Use Case in SOS31 in appendix B).
- The system shall allow users who are already logged-in to logout from the system (see Use Case SOS32 in appendix B).

1.2.2 Non-Functional Requirements

Below is a summary of the non-functional requirements of the SOS system. The expected requirements for each Use Case have been collated into general system-wide requirements. A more detailed description of the non-functional requirements can be found in each use case in Appendix B.

1.2.2.1 Usability

In general, no training or special knowledge is required to use any of the implemented functionalities. For each user, a tutorial or help frame should be provided to guide new users. Users should take at most 10 minutes to find and use each of the functionalities provided by SOS.

1.2.2.2 Reliability

In general, a mean time to failure between 1 and 5% monthly is acceptable. Availability is affected by two downtimes, one for login back up, 30 minutes every 24-hour period, and another for maintenance, 1 hour in a 2 weeks period.

1.2.2.3 Performance

Privilege checks should be done within 2 seconds. The system should be able to handle 20 privilege checks in 1 minute. Each individual form and request should be sent, processed, and saved within at most 10 seconds. The system should be able to handle around 20 and 50 requests per minute.

1.2.2.4 Supportability

The whole system is supported by Chrome, Mozilla, and IE desktop and mobile browsers.

1.2.2.5 Implementation

The whole system is implemented using JS React for the front-end and Java-based software for the backend.

1.3 Development Methodology

The development of the Student Organization System (SOS) follows the Unified Software Development Process (USDP; Jacobson, Booch, & Rumbaugh, 1999). The USDP can be seen as defined by a set of interconnected models: (a) use case model, (b) analysis model, (c) design model, (d) deployment model, (e) implementation model, and (f) test model. Their relationships can be seen in Figure 1: The relationships between the models in the Unified Software Development Process (USDP).Figure 1.

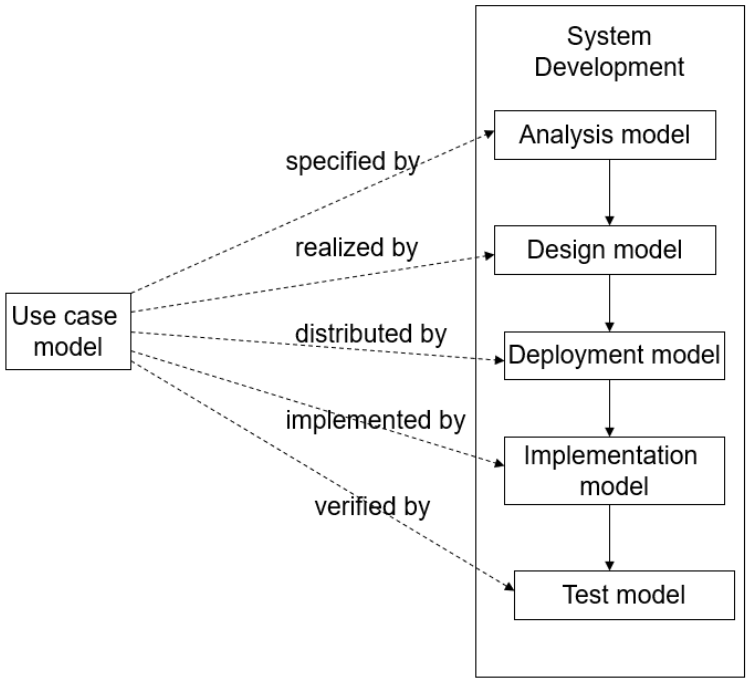

Figure 1: The relationships between the models in the Unified Software Development Process (USDP).

This document contains the third model, the design model described in Chapters 6-9. The design model gives a more detailed view of the system in the form of a set of interconnected subsystems, each containing classes and performing a discrete action. Sections 2.1 and 2.2 contain an overview of these subsystems in the form of a top-level UML Package Diagram. In addition, Section 3.1 contains the detailed designs of each of the subsystems in the form of simplified UML Class Diagrams. A more descriptive view of the UML Class Diagrams for the subsystem decompostion are found in Appendix C. A simplified version of the implementation model, is also presented in this document, in Section 2.3, Hardware and Software Mapping, which contains a UML Deployment Diagram of the SOS. The design and deployment models should provide a detailed description of the system structure without relying on implementation details and which could be ported to any desired platform with sufficient functionalities.

1.4 Definitions, Acronyms, and Abbreviations

Table 1: Definitions, Acronyms, and Abbreviation, contains a series of terms and acronyms used through this document. A more elaborate glossary can also be found in Section 6 of this document.

Term	***Meaning***
3TA	Three-Tier Architecture
API	Application Programming Interface
DB	Data Base (Data Storage)
DD	Design Document
FIU	Florida International University
FSD	Final Systems Document
N/A	Not Applicable
SOS	Student Organization System
SRD	Software Requirements Document
UML	Unified Modeling Language
USDP	Unified Software Design Process
V&V	Validation & Verification

Table 1: Definitions, Acronyms, and Abbreviation

1.5 Overview of the Document

This document is structured into chapters, each describes a different aspect of the project's system design. Chapter 2 discusses the software architecture concepts used to implement the SOS system. The software architecture involves the decomposition of the system into subsystems, hardware and software mapping, persistent data management, and security management. Chapter 3 wraps up all parts of detailed design including several different UML diagrams (package, class, object, state, and sequence diagrams) for every subsystem derived from the system decomposition. Chapter 4 goes through the glossary which defines the domain-specific keywords and terms used in this document. Chapter 5 is the approval page of the document which contains all of the SOS team members' signatures. Chapter 6 includes the references used in the document.

Finally, Chapter 7 contains five different appendixes (A-E). Appendix A includes the project's use case diagram, Appendix B includes detailed descriptions of each implemented use case in the project (total of 10). Moreover, Appendix C contains detailed subsystem class diagrams (total of 7), Appendix D describes the Java code for the subsystem classes, Appendix E contains diaries for every meeting held during this deliverable milestone.

2 Proposed Software Architecture

The following sections contain a top-level description of the architecture of the Student Organization System (SOS), including subsystems decomposition, as well as data management and security requirements. Section 2.1 contains a general overview of the system, including a general description of the architectural patterns used. Following that, Section 2.2 contains a subsystem decomposition for the SOS. Section 2.3 contains a UML Deployment Diagram showing the hardware and software mapping expected for the system. Section 2.4 contains the requirements and schema used for persistent data in the system. Finally, Section 2.5 contains the security requirements and schema for the system.

2.1 Overview

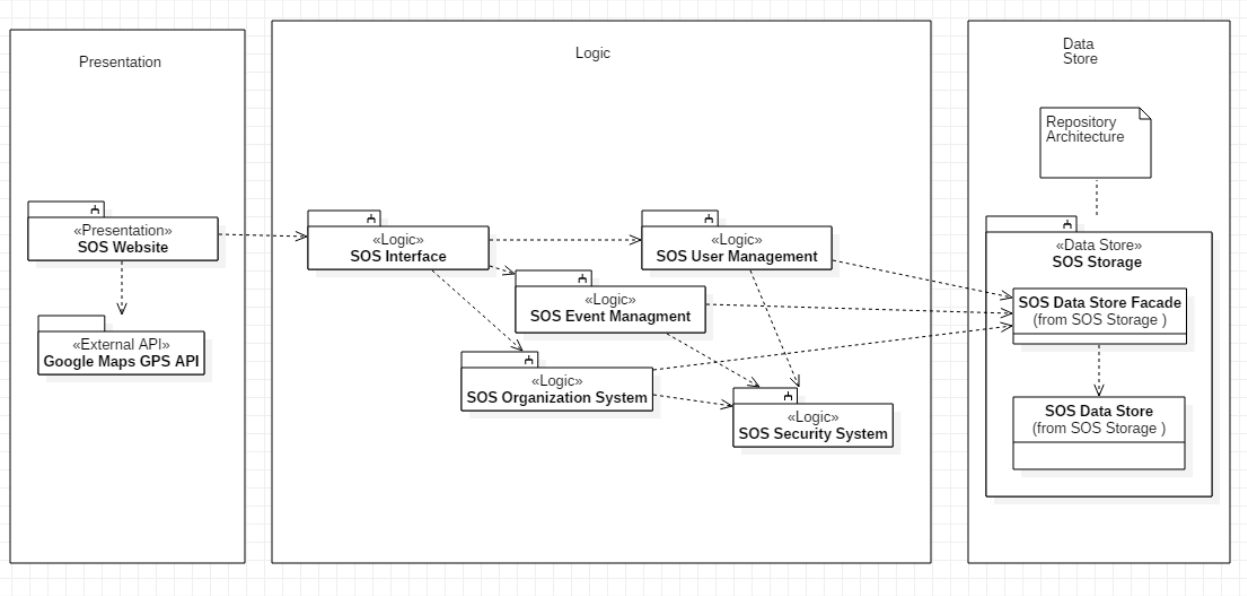

Figure 2: Class Diagram showing the top-level architecture of the Student Organization System.

The SOS system is implemented using a three-tier architecture (3TA). In a 3TA, systems components are divided along three layers: (a) an interface layer, which includes the objects that interact with the user, in the SOS's case, a front-end Website; (b) an application logic layer, which includes the control and entity objects implementing the system's logic, in the SOS's case, a back-end Java server; and (c) a storage layer, which contains, maintains, and retrieves the persistent objects found in SOS. The 3TA was chosen because it allows the SOS system to be divided into interchangeable layers which can be updated and maintained separately if their separate interfaces are maintained. Moreover, it allows each of the layers to be hosted in different systems, which matches the desired deployment structure of a front-end client, a back-end system, and a separated storage system (see Section 2.3 for a full deployment description). In addition, 3TA has superior performance for medium-to-high volume environment, which matches the expected volume that the SOS system would experience if deployed in its target environment (universities and other similar closed communities). The SOS system subdivides its structure into more than three

subsystems, but these are grouped into each of the three layers of a 3TA. This mapping is presented in the following section, Section 2.2.

Besides the 3TA, the SOS system also implements a repository architecture. In a repository architecture, several subsystems access and modify data from a single data structure (a repository) which mediates their interaction. This architecture is used in the storage layer. Because our primary architecture is 3TA, most of the subsystem interaction is not mediated by the repository, but instead by within-layer connections. However, some subsystems do interact with the repository when calling functions of the storage facade within the storage layer. This architecture was chosen because it serves as an efficient way to store a large amount of data and retrieve it from a single monolithic source. Moreover, it reduces the overheard of a transient data between software components.

The combination of these two architectures was chosen to meet the standards and expectations of the non-functional requirements of both performance and reliability, since both architectures ensure that the system will be responsive and quick to handle requires.

2.2 Subsystem Decomposition.

The following subsystems compose the Study Organization System:

- SOS Storage, which will act as a central node in the repository architecture where persistent data is stored, maintained, and retrieved. It alone is part of the storage layer in the 3TA. All use cases that interact with the data store use this subsystem.
- SOS Website, which represents the interface layer of the 3TA. It contains the objects which will present the SOS site that acts as the user interface. This will be done on each user's browser (front-end). All use cases will by default use this subsystem.
- SOS Interface, which acts as the server of the application which processes requests from the SOS website and create solution objects of the other subsystems that will resolve those requests and interact with the data store. This subsystem is a core part of the application logic layer of the 3TA. All use cases will by default use this subsystem.
- User Management, which contains all the system functions relating to Users, such as Registration (SOS22), Edit Profile (SOS07), and User Roles (SOS02, Grant Organizer Role). This subsystem is part of the application logic layer of the 3TA.
- Event Management, which contains all the system functions relating to events, such as Event Creation (SOS01), Attending Events (SOS04), Accessing Events by Location (SOS10), and Canceling Events (SOS17). This subsystem is part of the application logic layer of the 3TA.
- Organization Management, which contains all the system functions relating to Organizations, such as Granting Organizer Roles (SOS02), and Creating an Organization (SOS16). This subsystem is a part of the application logic layer of the 3TA.
- Security Management, which contains all the security-related functions, which mostly include password management and access control. These functions relate to User Roles (SOS02), Editing Profile Access (SOS07), Registration (SOS22), Login In (SOS31) and Out (SOS32). This subsystem is a part of the application logic layer of the 3TA.

- Google Maps GPS API, which represents an external API responsible for retrieving location coordinates for Events and Users. This is used for Creating Events (SOS01) and Accessing Events by Location (SOS10).

2.3 Hardware and Software Mapping

The hardware and software mapping for the SOS system can be seen in the UML Deployment diagram in Figure 3. The system uses three nodes, one web or mobile node for the client (front-end), a dedicated server for the SOS logic-layer (back-end) and a third dedicated server for the SOS data store layer. Alternatively, the two back-end layers could be unified into a single node.

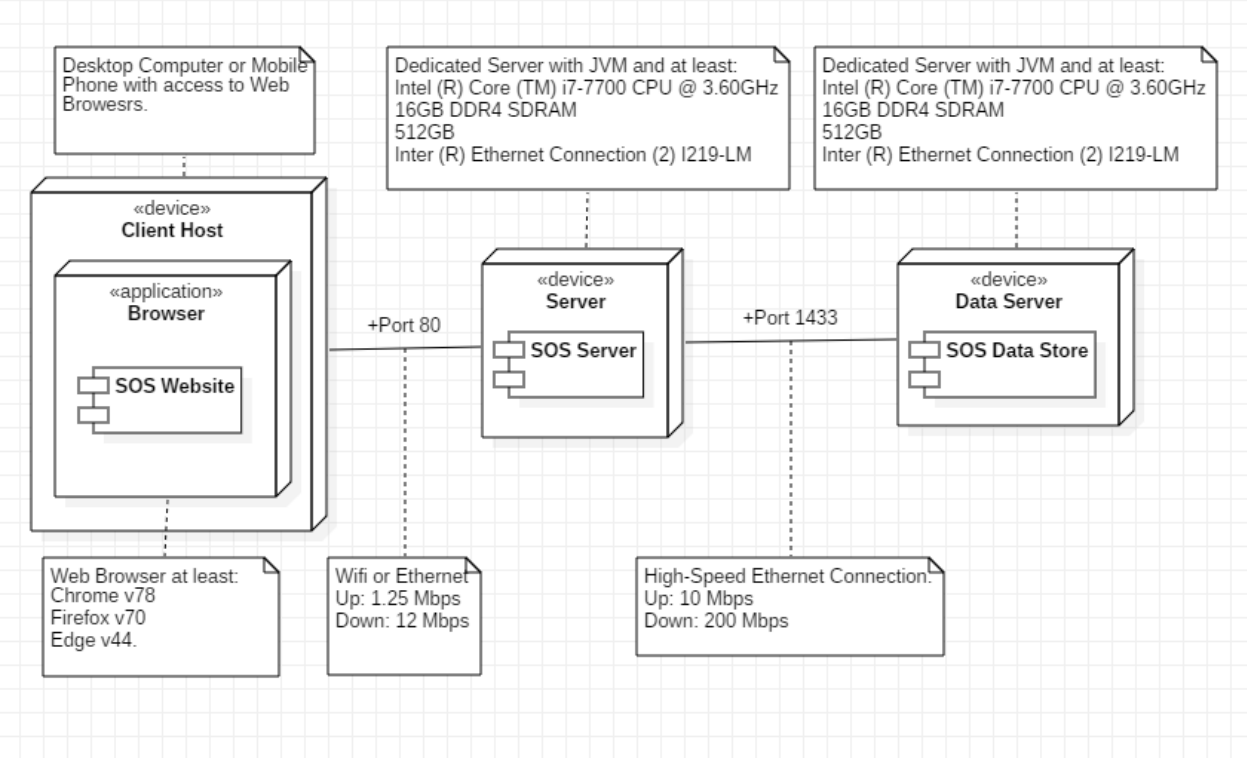

Figure 3: UML Deployment Diagram for the SOS system.

2.4 Persistent Data Management

The persistent entities for the SOS system, as well as the connections between them, are represented in the UML ER diagram in Figure 4.

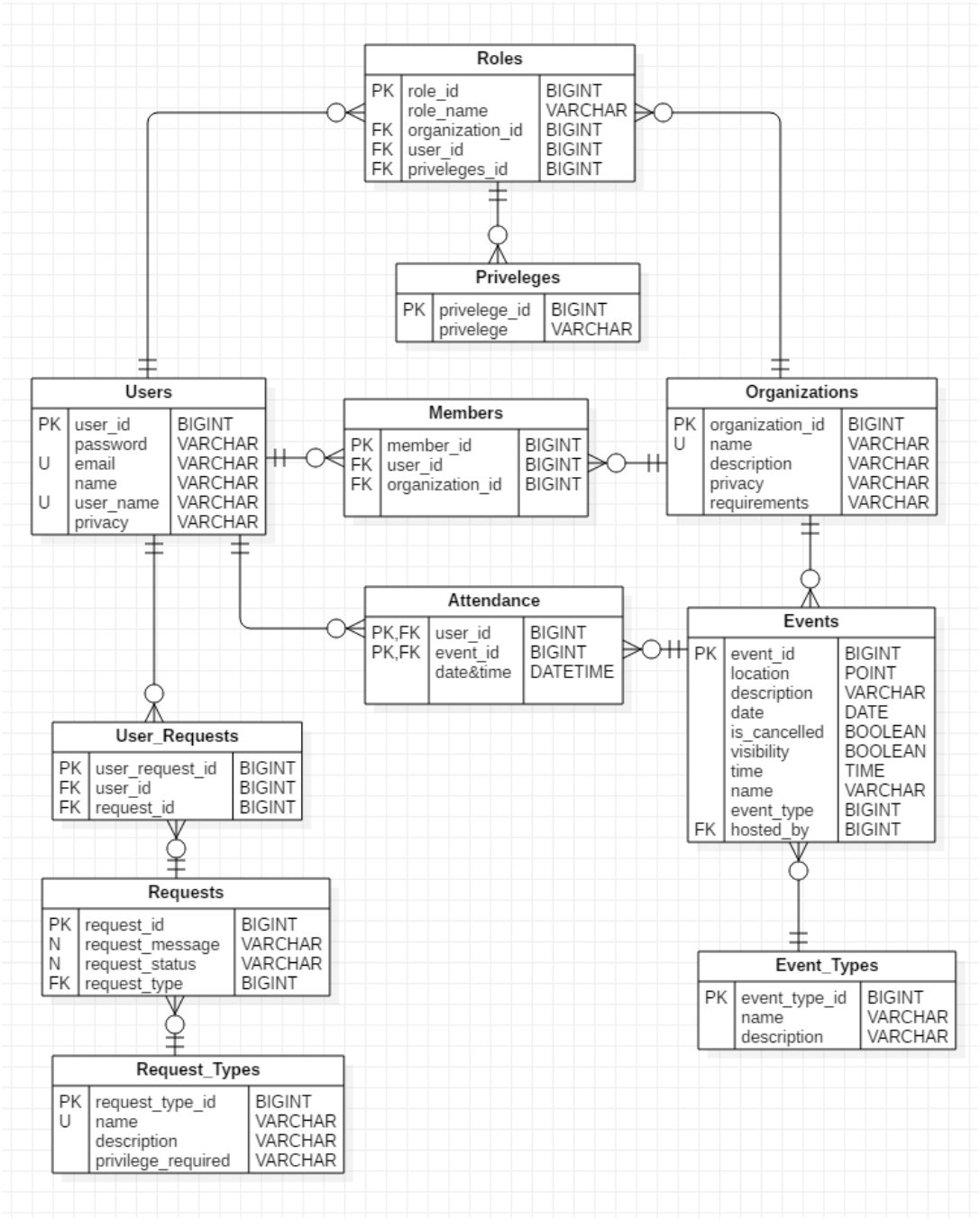

Figure 4: UML ER Diagram for the SOS System.

The diagram observers Third-Normal Form. The SOS system has the following tables:

- *Users*, which represent the user-defined accounts on the system. Users can be Members of Organizations, they can also have Roles (e.g., Organizer) in Organizations, they can attend Events, and they can make Requests on the system.
- *Members*, which is a link between Users and Organizations. A Member (which is an Actor in our system) is a User that belongs to an Organization.

- *Organizations*, which represent groups of Users in the system. Organizations have Members, which are Users that may or may not have privileges, and Organizers, which are Users which have Roles, with privileges. Organizations can host Events.
- *Roles*, which represents a set of privileges a User has in an Organization. A Role defines an Organizer (which is an Actor in our system).
- *Privileges*, which is a right a User might have with respect to an Organization. There are a set number or Privileges, which includes Create Event, Invite Users, Delete Organization, etc.
- *Events*, which represent real-life activities. Events are associated with their hosting Organization and can be Attended by Users. Events also have Types.
- *Event_Types*, which represent types of Events. There is a set number of accepted Types.
- *Attendance*, which is a link between Users and Events.
- *Request*, which is a request on the system by a User. They are kept for housekeeping and maintenance purpose.
- *User_Requests*, which is a link between Users and Requests.
- *Request_Types*, which represent types of Requests. There is a set number of accepted Types.

The descriptions for each field in Figure 4, as well as field size, data type, and format, can be seen in the data dictionary in Table 2.

entity name:	field name	field size	data type	Data Format	Example	Description
user	name	20 chars	string	----------	Rick Sanchez	First name of the user.
	user_name	20 chars	string	----------	TiredgeSnius68	The user's username.
	email	20 chars	string	username@domainname	Szechuan808@hotmail.com	The email address of the user.
	password	20 chars	string	----------	3o9t23bf4180rf87b2387	The encrypted password of the user.
	user_id	------	int	----------	1	A unique ID for the User
	privacy	20 chars	string	PRIVATE PUBLIC	PRIVATE	The privacy setting of the account
member	member_id	------	int	----------	1	A unique ID for the relation.
	user_id	------	Int	----------	1	Identifies a User
	organization_id	------	Int	----------	1	Identifies an Org.

Organi-zation	organization_id	------	int	----------	1	Identifies an Org.
	name	100 chars	string	----------	Club SOS	The name of the Organization
	description	500 chars	string	----------	A club for club-goers	The description of the Org.
	privacy	20 chars	string	PRIVATE PUBLIC	PUBLIC	The privacy setting of the Org.
	requirements	500 chars	string	----------	Be a FIU Student	The requirements for joining.
Roles	role_id	------	Int	----------	1	Identifies a Role.
	role_name	20 chars	string	----------	Owner	A given name for the Role.
	organization_id	------	Int	----------	1	Identifies an Org.
	user_id	------	Int	----------	1	Identifies a User.
	priviledges_id	------	Int	----------	1	Identifies a Priv.
Privi-ledge	priviledge_id	------	Int	----------	1	Identifies a Priv.
	privilege	20 chars	String	CREATE READ UPDATE DESTROY …	CREATE	A defined and immutable set of privileges which organizers might have in their Orgs
event	event_id	------	Int	----------	1	Identifies a Event
	location	------	Point	----------	FIU	A point to a location using the Google GPS API.
	description	500 chars	String	----------	1st Meeting	The description of the Event.
	date	------	Date	MM/DD/YY	10/10/19	Date of the Event
	is_cancelled	------	Bool	----------	FALSE	True if the Event is cancelled.
	visibility	------	Bool	----------	TRUE	True if the Event is visible.
	time	------	Time	HH:MM TM	10:30 AM	Time of the Event
	name	100 chars	String	----------	SOS Meeting	Name of the Event.
	event_type	------	Int	----------	1	Type of the Event.
	hosted_by	------	Int	----------	1	Identifies the hosting Org.
Event Types	event_type_id		Int	----------	1	Identifies the Event Type.
	name	20 chars	String	----------	Meeting	The name of the event type.
	description	100 chars	String	----------	A get-together.	The description of the event type.

Atten-dance	user_id	------	Int	----------	1	Identifies a User
	event_id	------	Int	----------	1	Identifies a Event
	datetime	------	Date-time	----------	10/10/19, 10:30 AM	The date and time of the attendance.
Request	request_id	------	Int	----------	1	Identifies a request.
	request_messa-ge	500 chars	String	----------	Add General Meeting Event.	The message of the request.
	request_status	100 chars	String	----------	Valid	The resolution of the request.
	request_type	------	Int	----------	1	Identifies a Request Type.
User Request	user_request_id	------	Int	----------	1	Identifies a user-request link.
	user_id	------	Int	----------	1	Identifies a User.
	request_id	------	Int	----------	1	Identifies a Request.
Request Types	request_type_id	------	Int	----------	1	Identifies a Request Type.
	name	100 chars	String	----------	Add Event	The name for the type of request.
	description	500 chars	String	----------	Adds and Event from an Org.	The description for this type of request.
	privilege_req	100 chars	String	----------	Event Creation.	The required privilege for this request.

Table 2: Data Dictionary for the SOS Persistent Data.

2.5 Security Management

The SOS uses two core security mechanism, Password Management, which is described in Section 2.5.1, and Access Management, which is described in Section 2.5.2. In both cases, the relevant classes are implemented in the Security Subsystem (see Section 3.1.7). Besides these two functionalities, other systems are also used such as API Keys for the Google Maps GPS API and Encryption for network sharing of important data.

2.5.1 Password Management.

The goal of the Password Management policies is to ensure authenticity of the Users logged onto the SOS, and to ensure that changes issued by those User's accounts are committed by them and not by third parties who have gained access to their account. In order to do this, accounts must be locked behind passwords which only the real User should know. Hence, these passwords should be protected tightly by the SOS so no third-party gain access to them.

To ensure the safety of the password, the system encrypts it at the client side and shares it through the network encrypted. The encryption method used is public-private key encryption (RSA): when a session starts, the client receives a public key from the system, which it can use to encrypt the password. This ciphertext is then sent over the network to the back-end which decrypts it using its corresponding private key. In order to avoid storing real passwords in the back-end, the front end will hash the plaintext password with a salt value to create a unique hash. The hash and the salt will be sent to the backend and stored in leu of the actual password. To make things simple, the salt value will be the account's username.

In order to ensure hard-to-crack password, the following policy will be enforced:

- Passwords must be at least 6 characters long.
- Passwords must have at least 1 uppercase character.
- Passwords must have at least 1 number or special character.

2.5.2 Access Management

The goal of the Access Management policies is to ensure authorization of the actions that known Users are doing within the system, i.e., to ensure that Users can only do the actions that they can perform. In the SOS's case, the main actions involve exclusively creating, reading, updating, and destroying persistent data object such as Events, Organizations, and Users (i.e., Accounts). Because of this, a simple view of the access management policy can be represented using an Access Matrix on these objects, as is seen in Table 3.

		Data Types			
		Events	Organizations	Users Accounts	Roles
Actors	Member (Non-Owner)	R	R	R	R
	Organizer (Non-Owner)	R	R	R	R
	Member (Owner)	*Not Applicable*	*Not Applicable*	CRUD	*Not Applicable*
	Organizer (Owner)	CRUD	CRUD	CRUD	CRUD

Table 3: Access Matrix for the SOS System. Uses the CRUD mnemonics: Create, Read, Update, and Destroy. Note that Users cannot own Events, Organizations, or Roles, so the CRUD is not applicable to those relations.

The access policy, especially with regards to Organizations, is based on the notion of Privileges, which are specific permissions which Users have with regards to system functions. For example, a User might have a "Create Event" privilege in a given Organization, which lets them create new Event objects hosted by that Organization. Note that the distinction between our two actors, Members and Organizers is effectuated within our system exclusively by means of privileges: Members are Users linked to Organizations while Organizers are Members which also have Roles assigned to them which give them Privileges on that Organization.

3 Detailed Design

The following sections contain a detailed description of the Student Organization System (SOS) in the form of UML package, class, state and sequence diagrams. Section 3.1 contains an overview of the system showing the minimal class diagram for each of the subsystem as well as a short description of each class depicted in those diagrams. Following that, Section 3.2 contains a state machine for the SOS in the form of a UML State Chart Diagram. Section 3.3 contains the object interactions for each of the implemented use cases of the SOS. Finally, Section 3.4 contains a detailed description of each class of the implemented subsystems, as well as OCL constrains for the control object on each subsystem.

3.1 Overview

Each of the following sections contains a minimal UML Class Diagram for each of the subsystems of the SOS. The subsystem decomposition of the SOS can be seen in Section 2.2. For each minimal Class Diagram, a complete equivalent diagram with attributes and operations can be found in Appendix C.

Note that each of the minimal class diagrams also contain the non-subsystem packages showing the relationships to classes on other subsystems.

3.1.1 SOS Website

The minimal class diagram for the SOS Website subsystem can be seen in Figure 5. A full equivalent class diagram can be found in Appendix C.

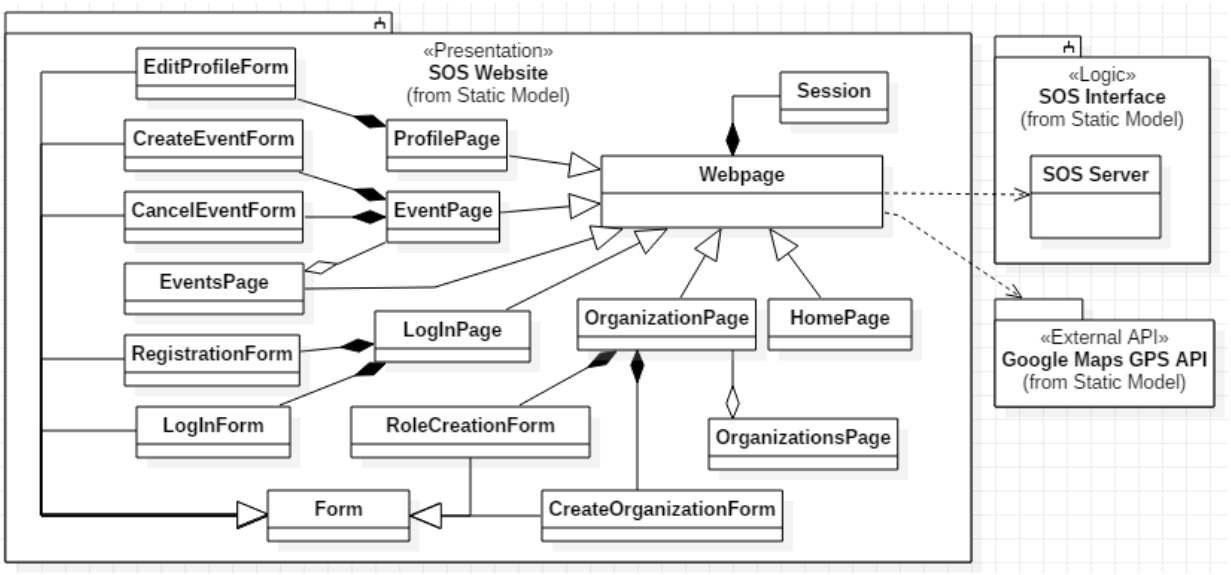

Figure 5: Minimal Class Diagram for SOS Website subsystem.

The following classes belong to this subsystem:

- Webpage, which is the core website class of the system, and the only one that interacts with the backend. Most other classes inherit from this one. The following classes extend Webpage with some specialized data:

- ProfilePage, which contains the User profile.
- EventPage, which contains Event data.
- EventsPage, which contains a list of Events.
- OrganizationPage, which contains Organization Data.
- OrganizationsPage, which contains a list of Organizations.
- HomePage, which contains the home page of the system.

- Form, which is the parent class for a series of input forms in the front end. These are:
  - LogInForm, which is the form for User Login.
  - RegistrationForm, which is the form for new User Registration.
  - CreateEventForm, which is the form for creating an Event.
  - EditProfileForm, which is the form for editing a User profile.
  - CreateOrganizationForm, which is the form for new Organization Creation.
  - RoleCreationForm, which is the form for new Role Creation.

3.1.2 SOS Interface

The minimal class diagram for the SOS Controller subsystem can be seen in Figure 6. A full equivalent class diagram can be found in Appendix C.

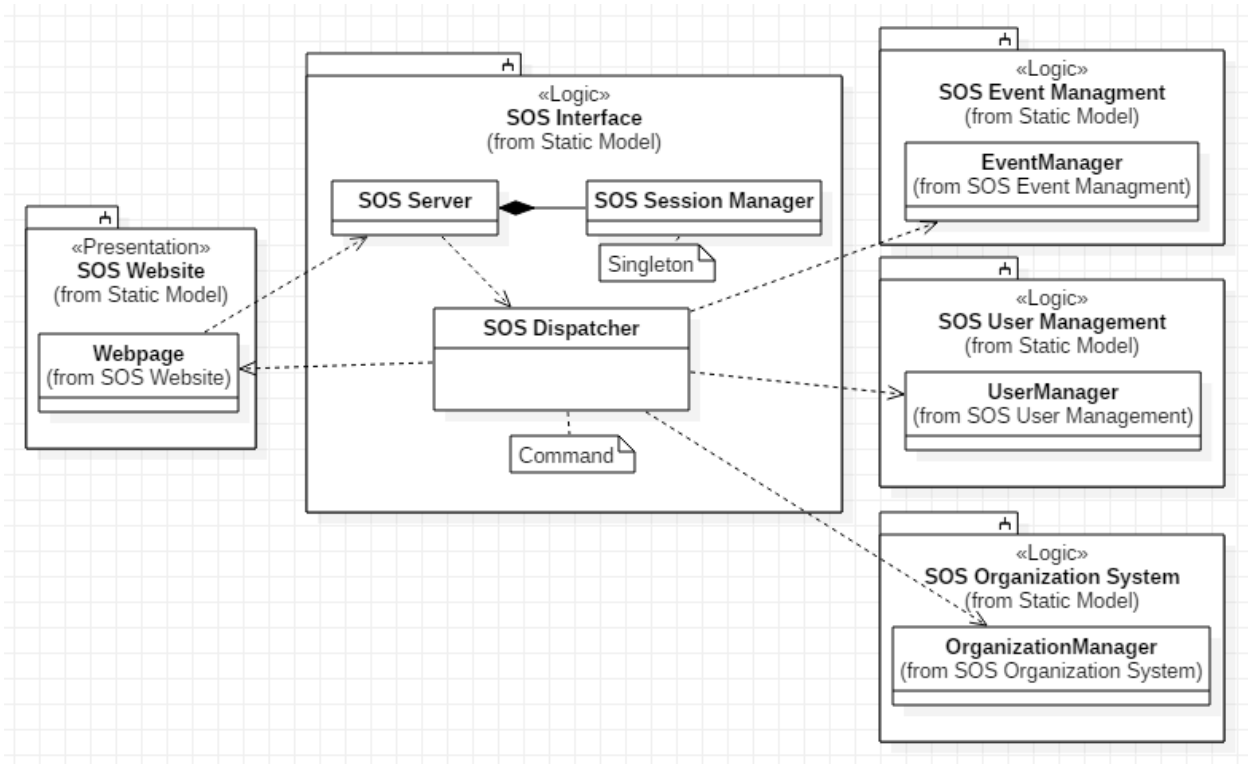

Figure 6: Minimal Class Diagram for SOS Interface.

- The following classes belong to this subsystem:
- SOS Server, which the main server instance of the system.
- SOS Session Manager, which has functions relating to system sessions.

- SOS Dispatcher, which propagates front-end request to their specific target controllers.

3.1.3 User Management

The minimal class diagram for the User Management subsystem can be seen in Figure 7. A full equivalent class diagram can be found in Appendix C.

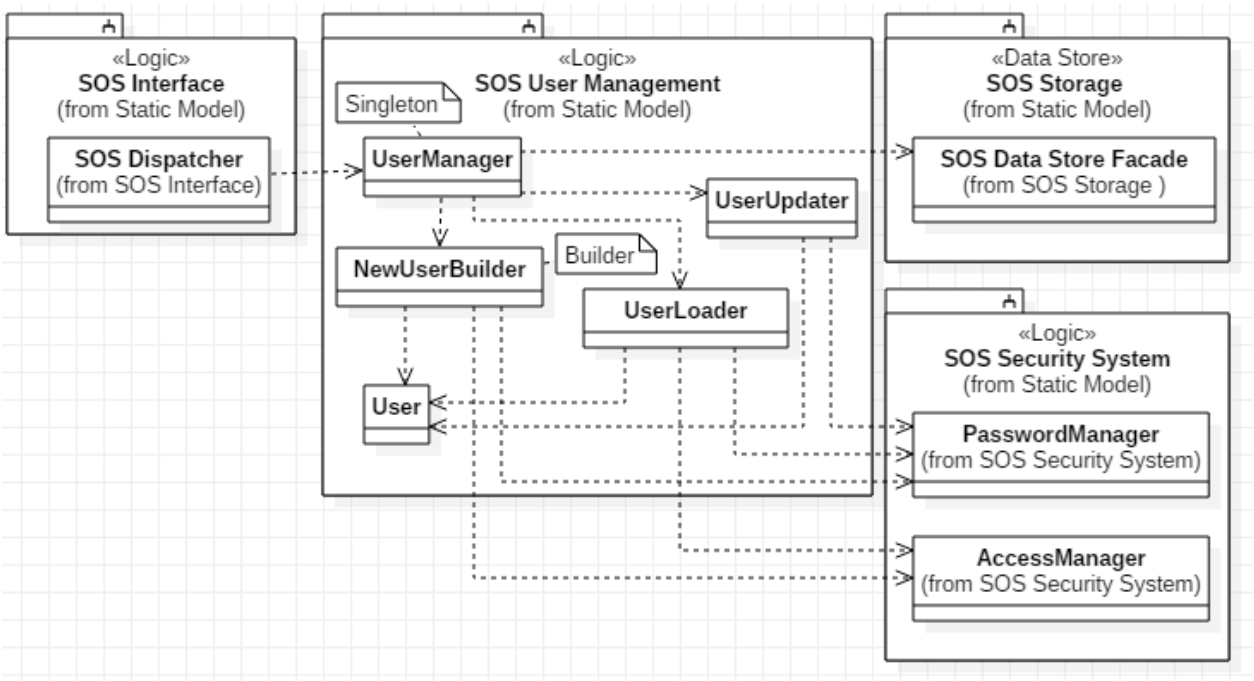

Figure 7: Minimal Class Diagram for SOS User Management.

The following classes belong to this subsystem:

- UserManager, which is a Singleton which manages all the User functions.
- NewUserBuilder, which is a Builder which creates new User objects.
- UserLoader, which is a class which creates a User object from a User database object.
- UserUpdater, which is a class which deal with User modifications.
- User, which is a run-time representation of a User persistent object.

3.1.4 Event Management

The minimal class diagram for the Event Management subsystem can be seen in Figure 8. A full equivalent class diagram can be found in Appendix C.

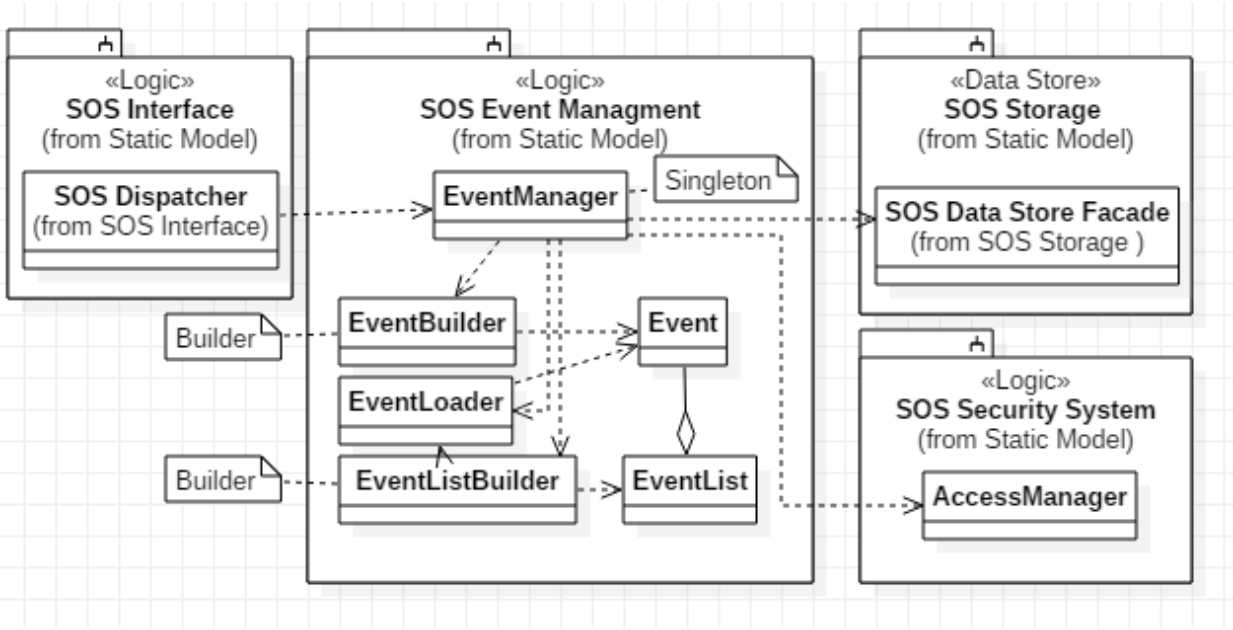

Figure 8: Minimal Class Diagram for SOS Event Management

The following classes belong to this subsystem:

- EventManager, which is a Singleton which manages all the Event functions.
- EventBuilder, which is a Builder which creates new Event objects.
- EventLoader, which is a class which creates an Event object from an Event database object.
- EventListBuilder, which is a Builder which creates new EventList objects.
- Event, which is a run-time representation of an Event database object.
- EventList, which is a class that aggregates Events.

3.1.5 Organization Management

The minimal class diagram for the Organization Management subsystem can be seen in Figure 9. A full equivalent class diagram can be found in Appendix C.

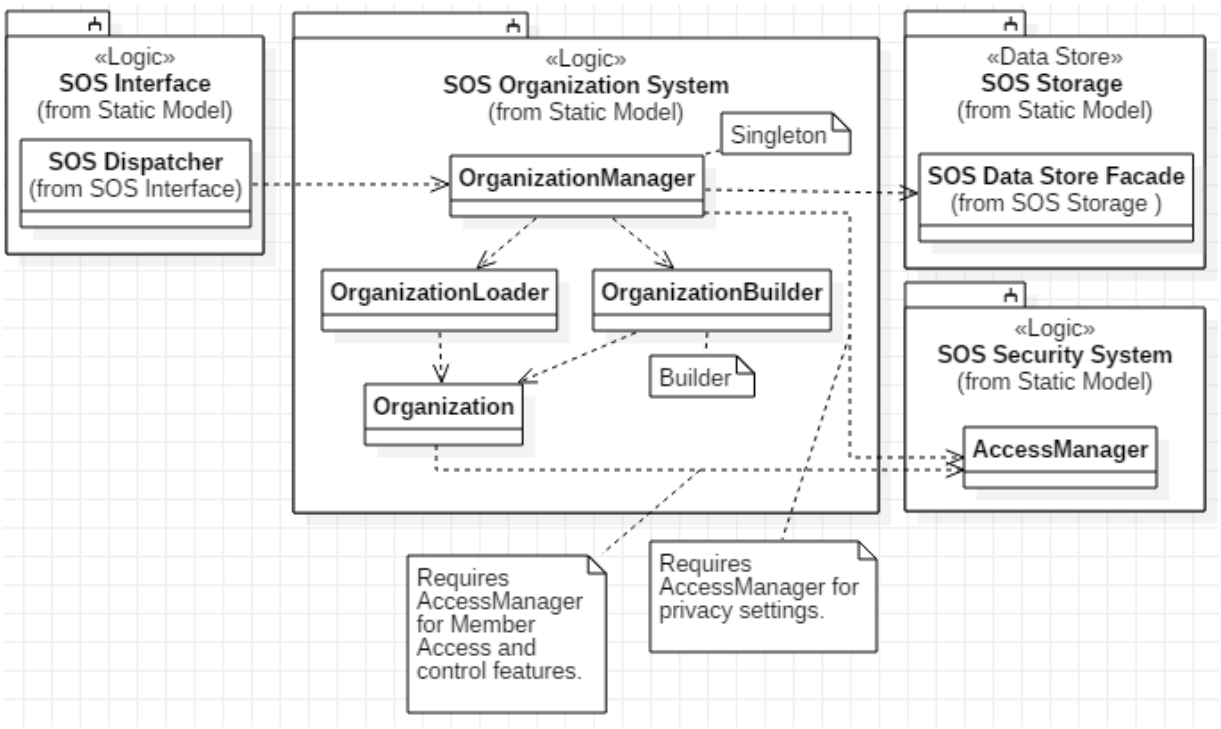

Figure 9: Minimal Class Diagram for SOS Organization Management

The following classes belong to this subsystem:

- OrganizationManager, which is a Singleton which manages all the Organization functions.
- OrganizationBuilder, which is a Builder which creates new Organization objects.
- OrganizationLoader, which is a class which creates an Organization object from an Organization database object.
- Organization, which is a class which deal with Organization modifications.

3.1.6 Security Management

The minimal class diagram for the Security Management subsystem can be seen in Figure 10. A full equivalent class diagram can be found in Appendix C.

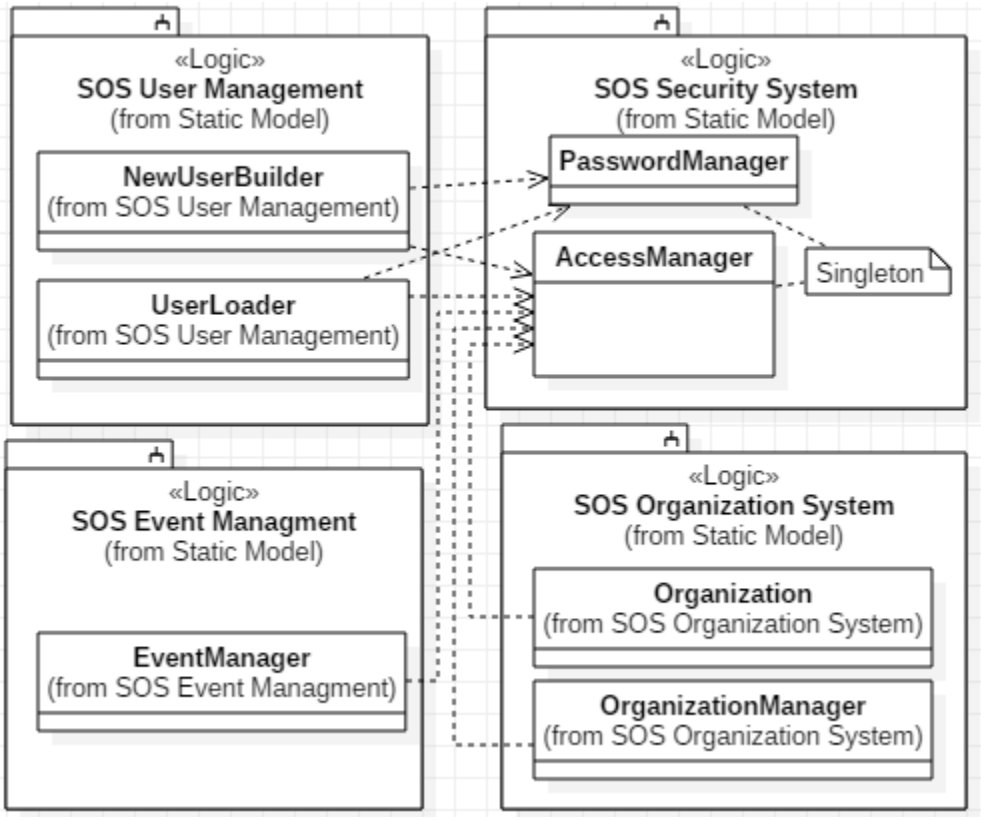

Figure 10: Minimal Class Diagram for SOS Security.

The following classes belong to this subsystem:

- PasswordManager, which is a Singleton dealing with password control actions.
- AccessManager, which is a Singleton dealing with access control actions.

3.1.7 SOS Storage

The minimal class diagram for the SOS Storage subsystem can be seen in Figure 11. A full equivalent class diagram can be found in Appendix C.

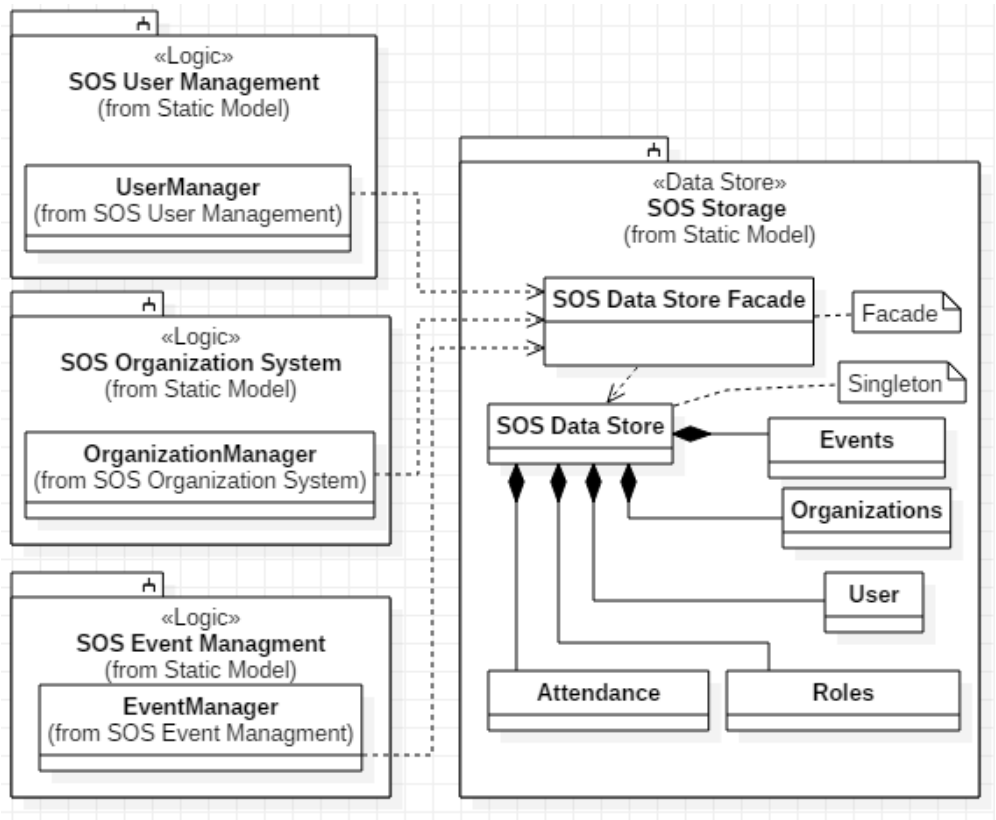

Figure 11: Minimal Class Diagram for Data Store subsystem.

The following classes belong to this subsystem:

- SOS Data Store Façade, which is the interface for the SOS Storage subsystem.
- SOS Data Store, which is the actual database implementation of the SOS system.
- Events, which is the Events table (see Section 2.4).
- Organizations, which is the Organizations table (see Section 2.4).
- User, which is the Users table (see Section 2.4).
- Roles, which is the Roles table (see Section 2.4).
- Attendance, which is the Attendance table (see Section 2.4).

3.1.8 Google Maps GPS API

The Google Maps GPS API does not have a class diagram as external it is just a software module imported into the front-end website code.

3.2 State Machine

The state machine for the SOS is shown in Figure 12. It provides a top-level, general description of the sequence of actions that our system takes, starting with user interactions through the Webpage, to processing and dispatching requests by the SOS Server, to executing those requests in the individual control objects of each subsystem and their interaction with the SOS Storage.

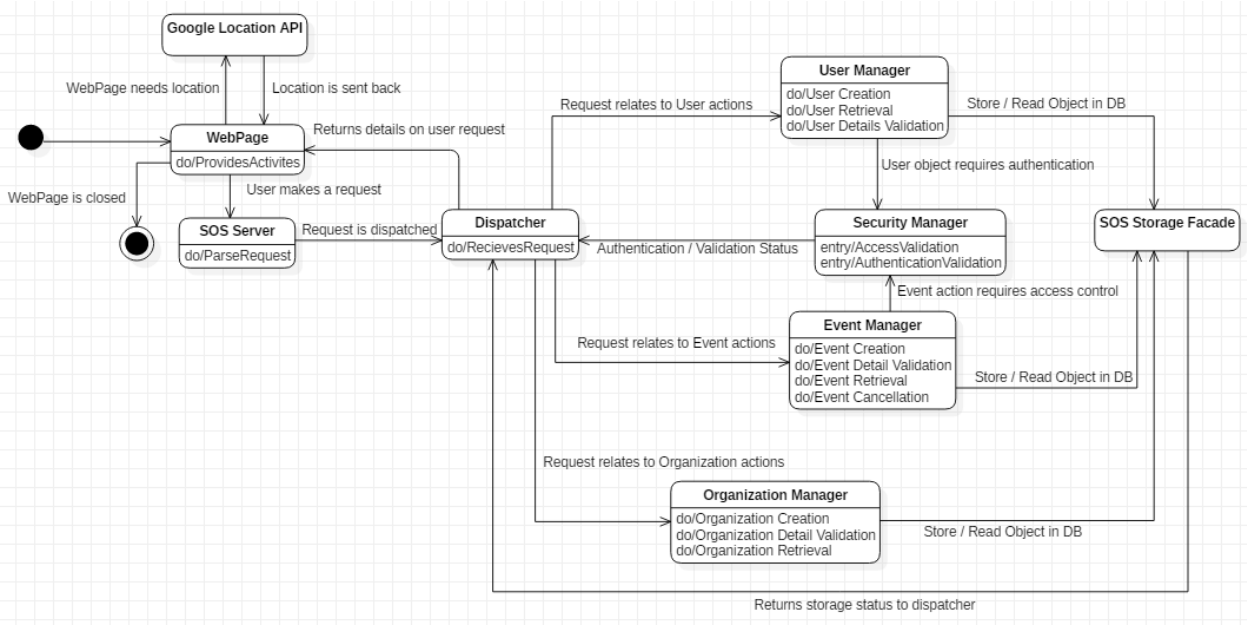

Figure 12: State Machine for the Student Organization System.

3.3 Object Interaction

Each of the following sections contain a sequence diagram detailing the object interactions for each of the implemented Use Cases for the SOS system. Each sequence diagram contains the interactions between the Actors, the core control objects, and key solution objects of the relevant subsystems which implement the functionality of the Use Case.

3.3.1 Sequence Diagram for SOS01 – Create an Event

The sequence diagram in Figure 13 corresponds to the Use Case in Appendix B, Section 7.2.1.

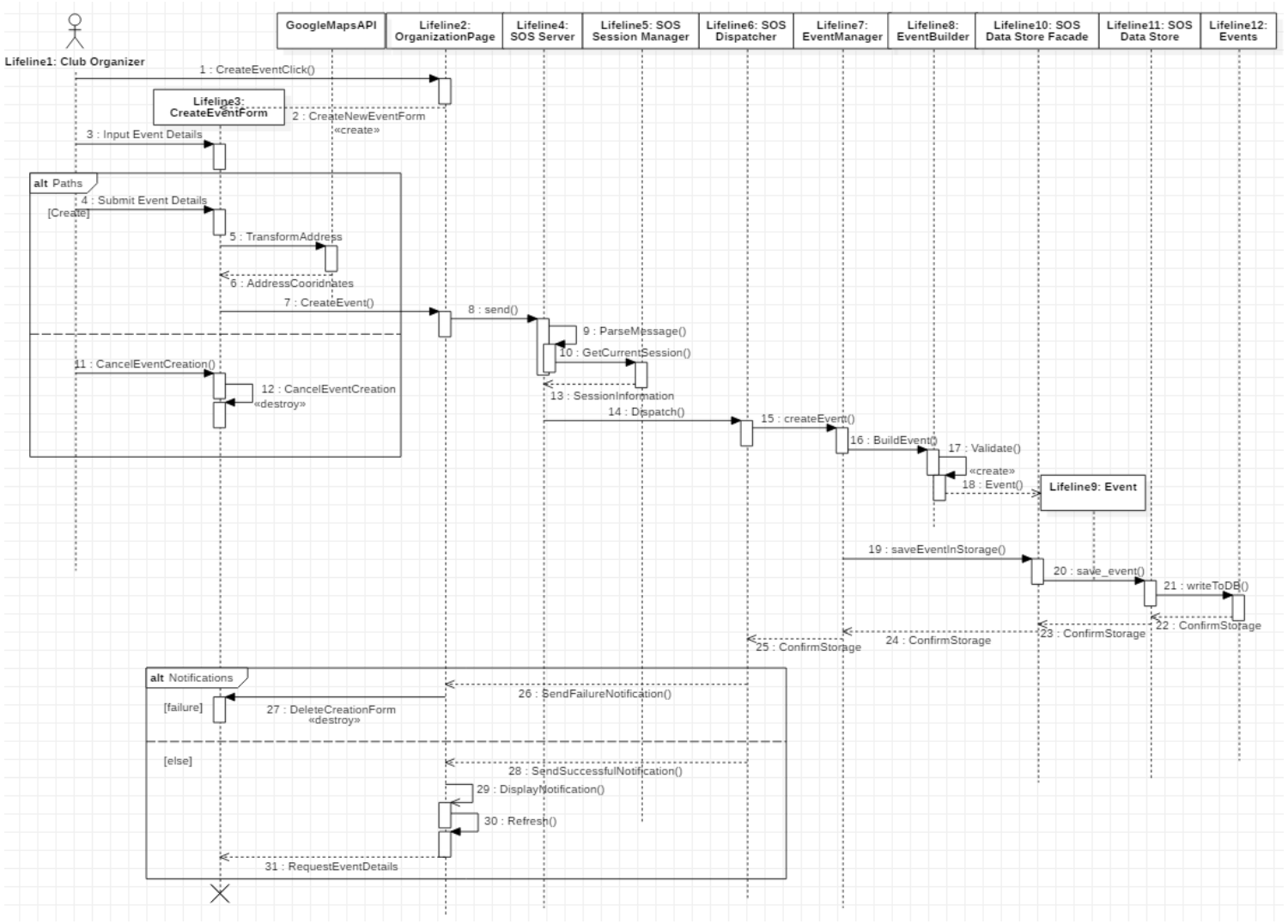

Figure 13: Sequence Diagram for SOS01 - Create an Event

3.3.2 Sequence Diagram for SOS04 – Attending an Event.

The sequence diagram in Figure 14 corresponds to the Use Case in Appendix B, Section 7.2.2.

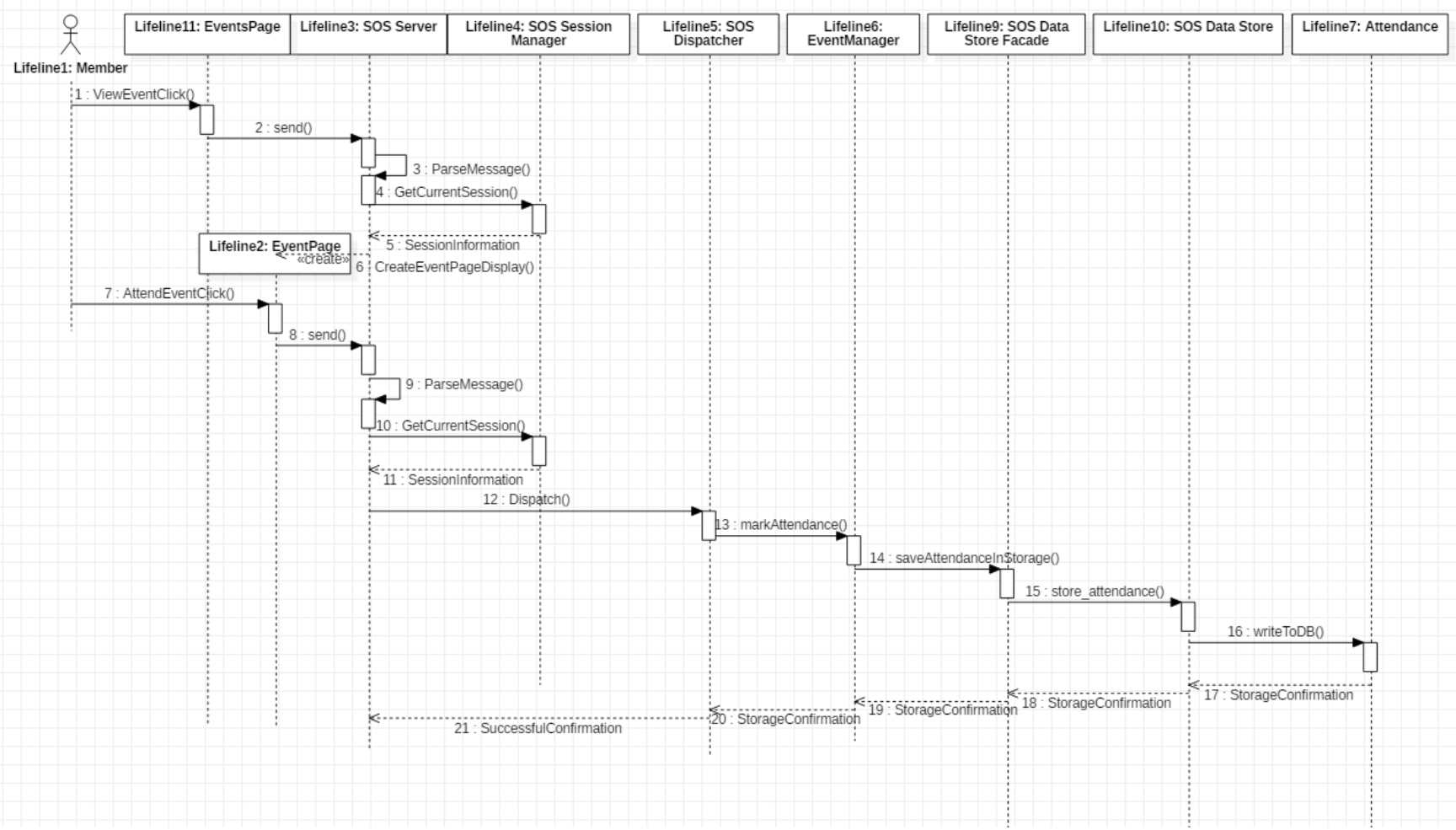

Figure 14: Sequence Diagram for SOS04 – Attending an Event.

3.3.3 Sequence Diagram for SOS02 – Grant Organizer Role

The sequence diagram in Figure 15 corresponds to the Use Case in Appendix B, Section 7.2.3.

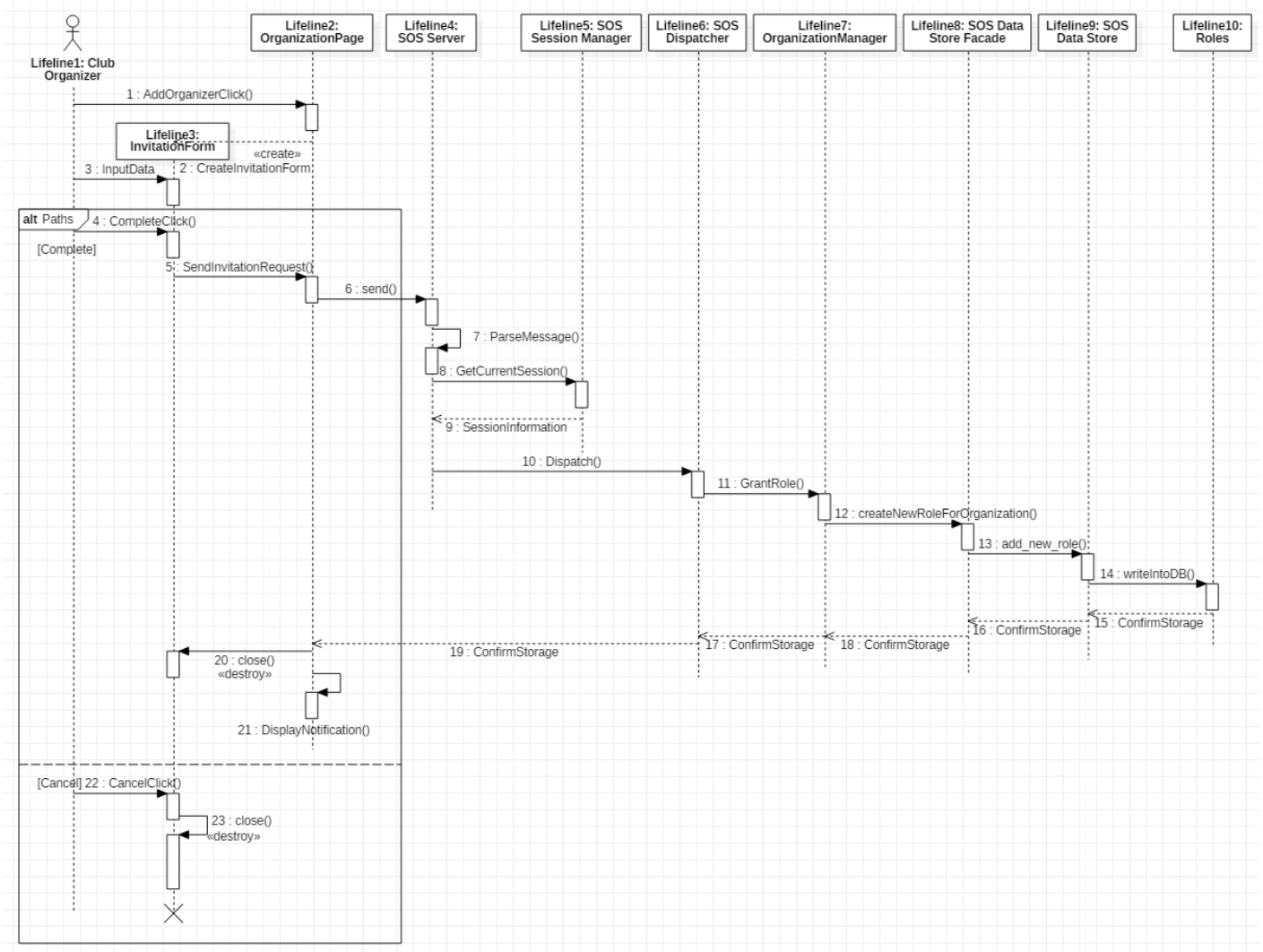

Figure 15: Sequence Diagram for SOS02 – Grant Organizer Role.

3.3.4 Sequence Diagram for SOS07 – Edit Profile

The sequence diagram in Figure 16 corresponds to the Use Case in Appendix B, Section 7.2.4.

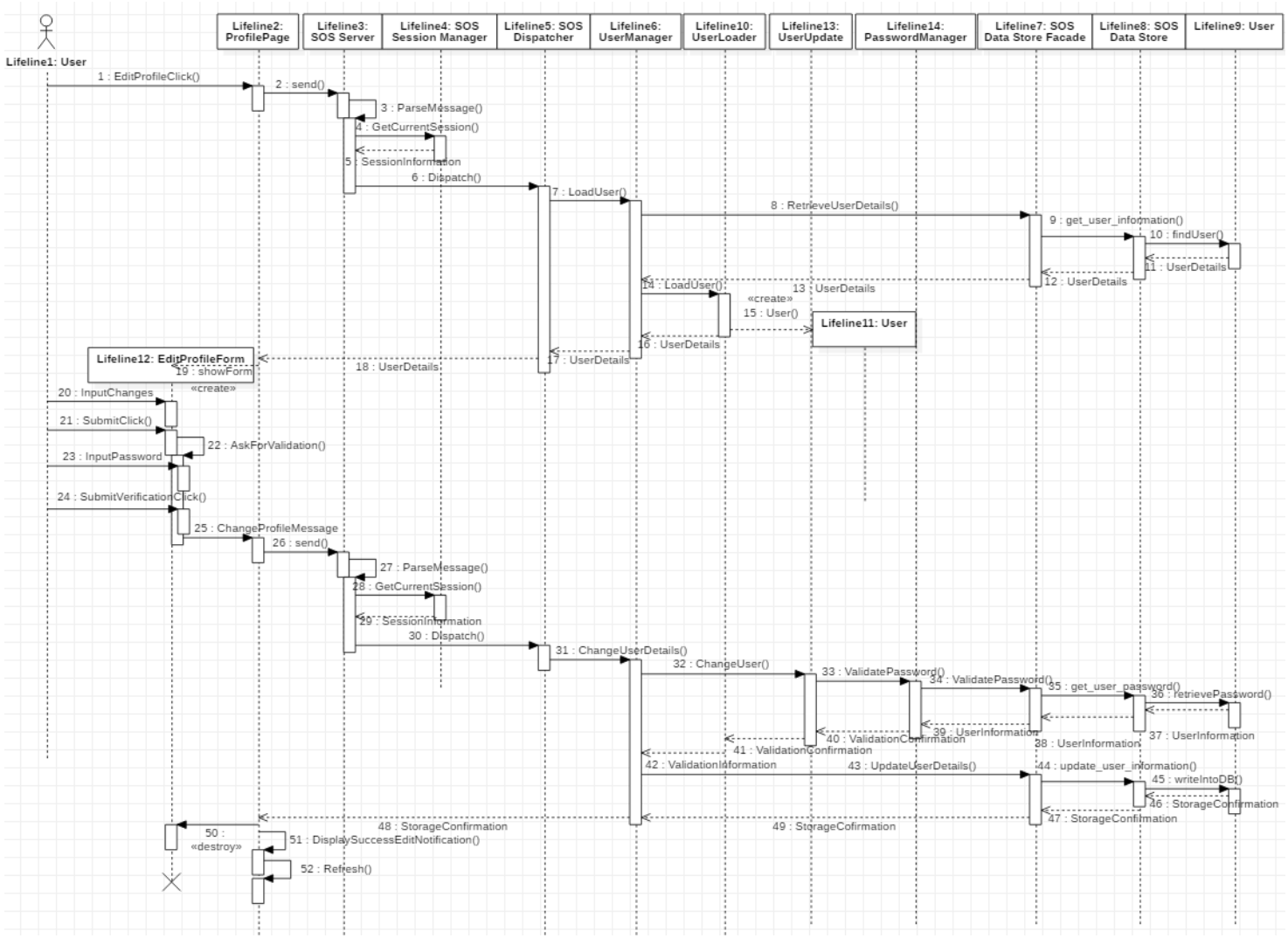

Figure 16: Sequence Diagram for SOS07 – Edit Profile.

3.3.5 Sequence Diagram for SOS16 – Create Organization

The sequence diagram in Figure 17 corresponds to the Use Case in Appendix B, Section 7.2.5.

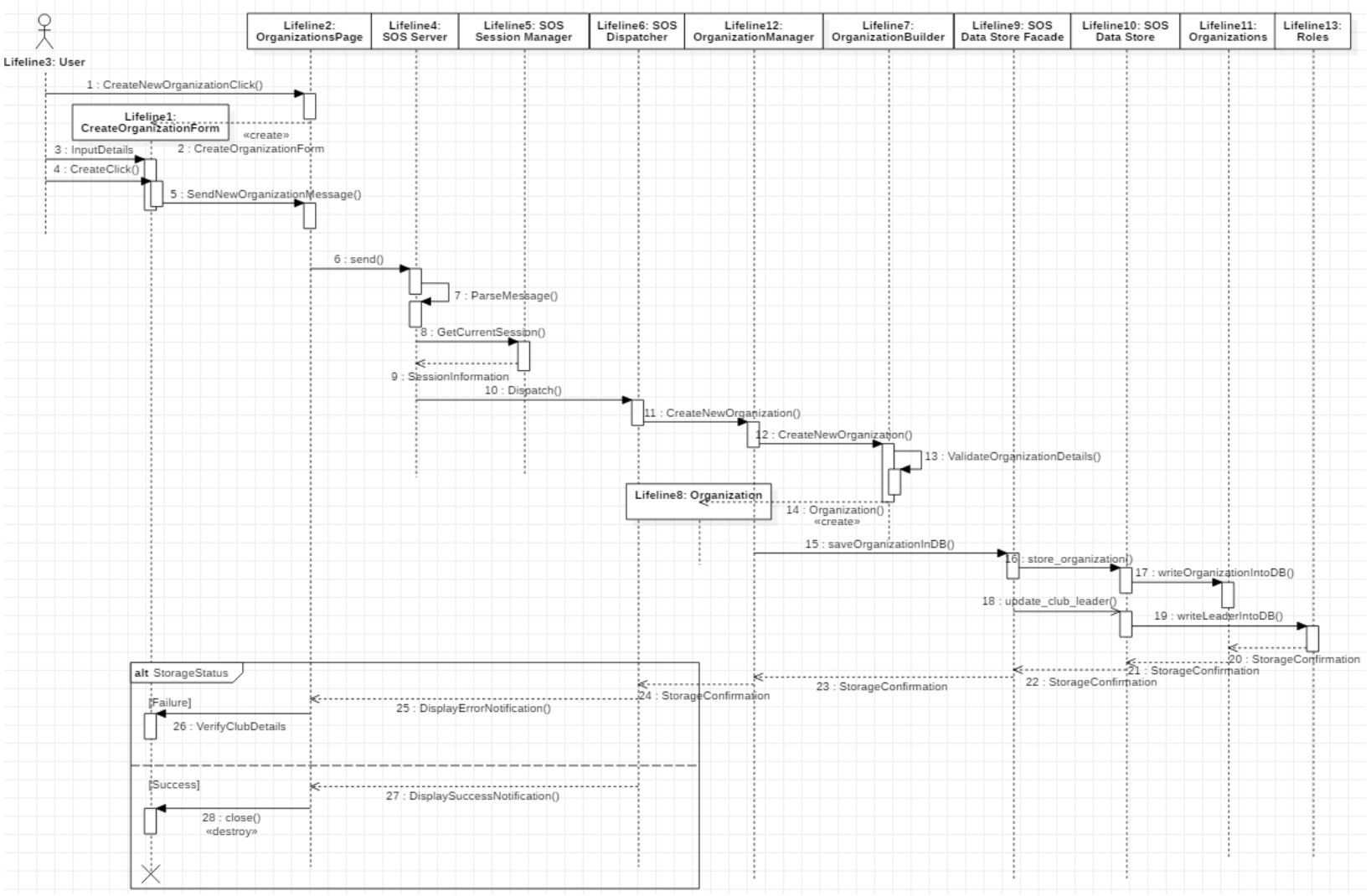

Figure 17: Sequence Diagram for SOS16 – Create Organization.

3.3.6 Sequence Diagram for SOS17 – Cancel an Event

The sequence diagram in Figure 18 corresponds to the Use Case in Appendix B, Section 7.2.6.

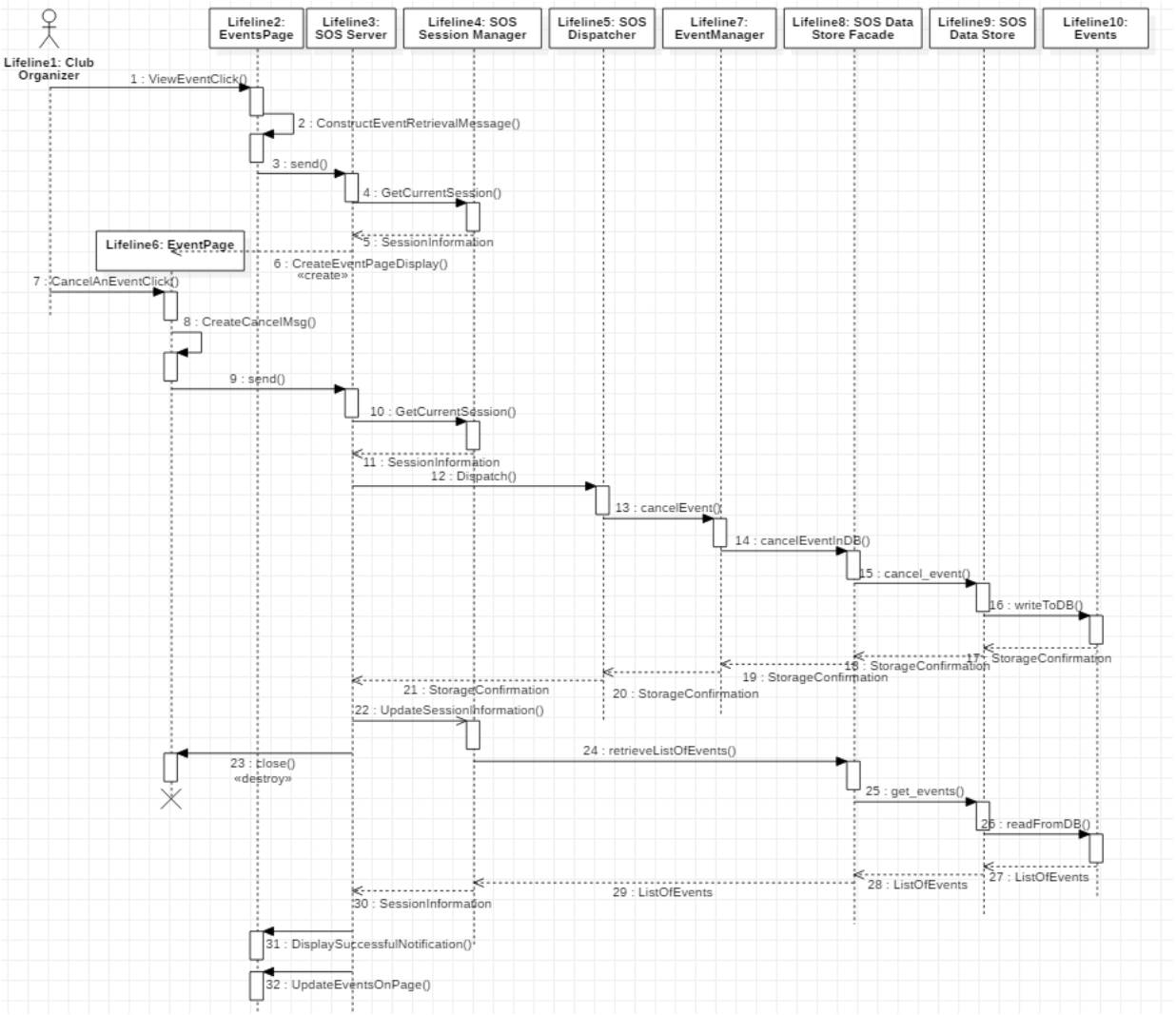

Figure 18: Sequence Diagram for SOS17 – Cancel an Event.

3.3.7 Sequence Diagram for SOS22 – Registration

The sequence diagram in Figure 19 corresponds to the Use Case in Appendix B, Section 7.2.7.

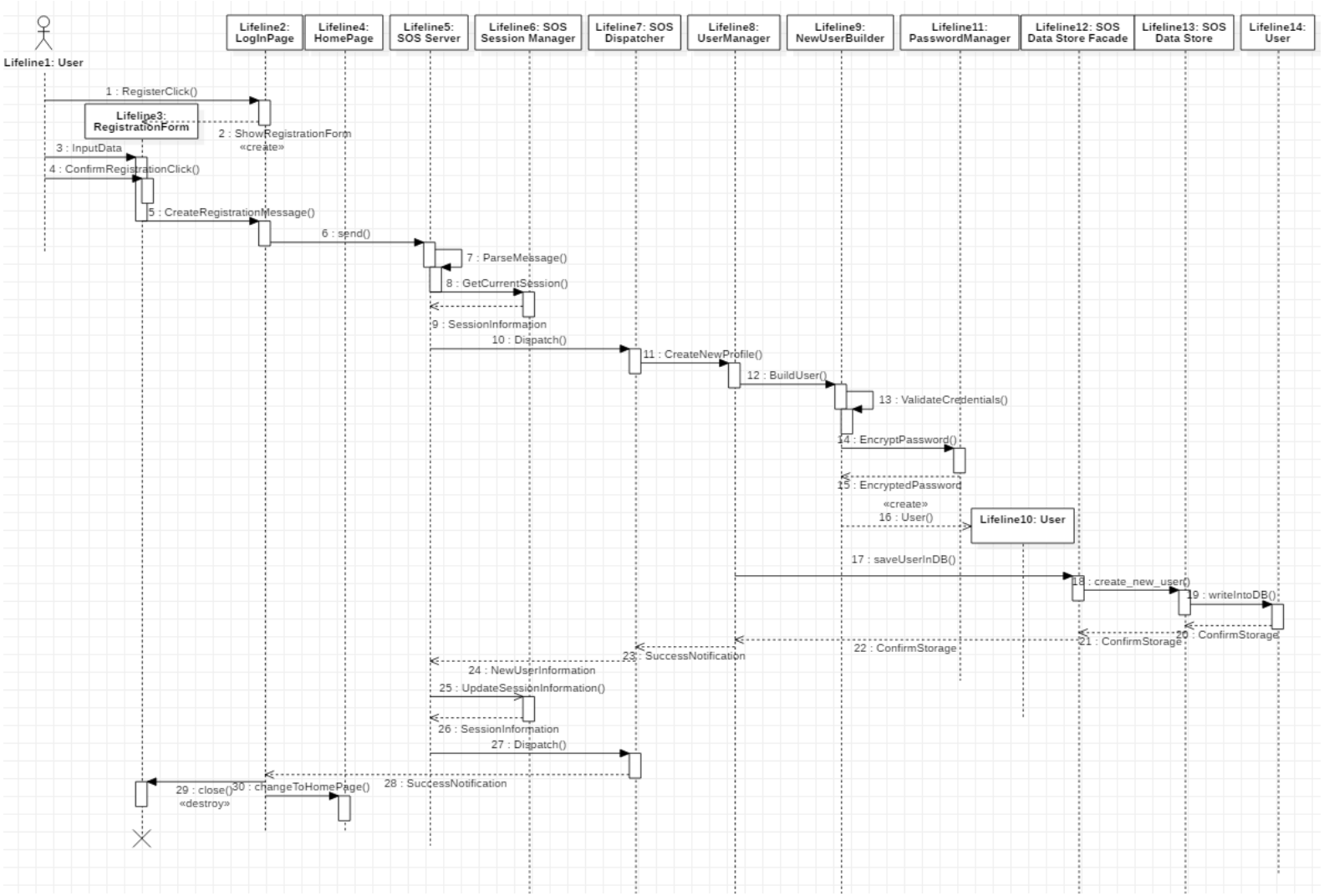

Figure 19: Sequence Diagram for SOS22 – Registration.

3.3.8 Sequence Diagram for SOS10 – Access an Event by Location

The sequence diagram in Figure 20 corresponds to the Use Case in Appendix B, Section 7.2.8.

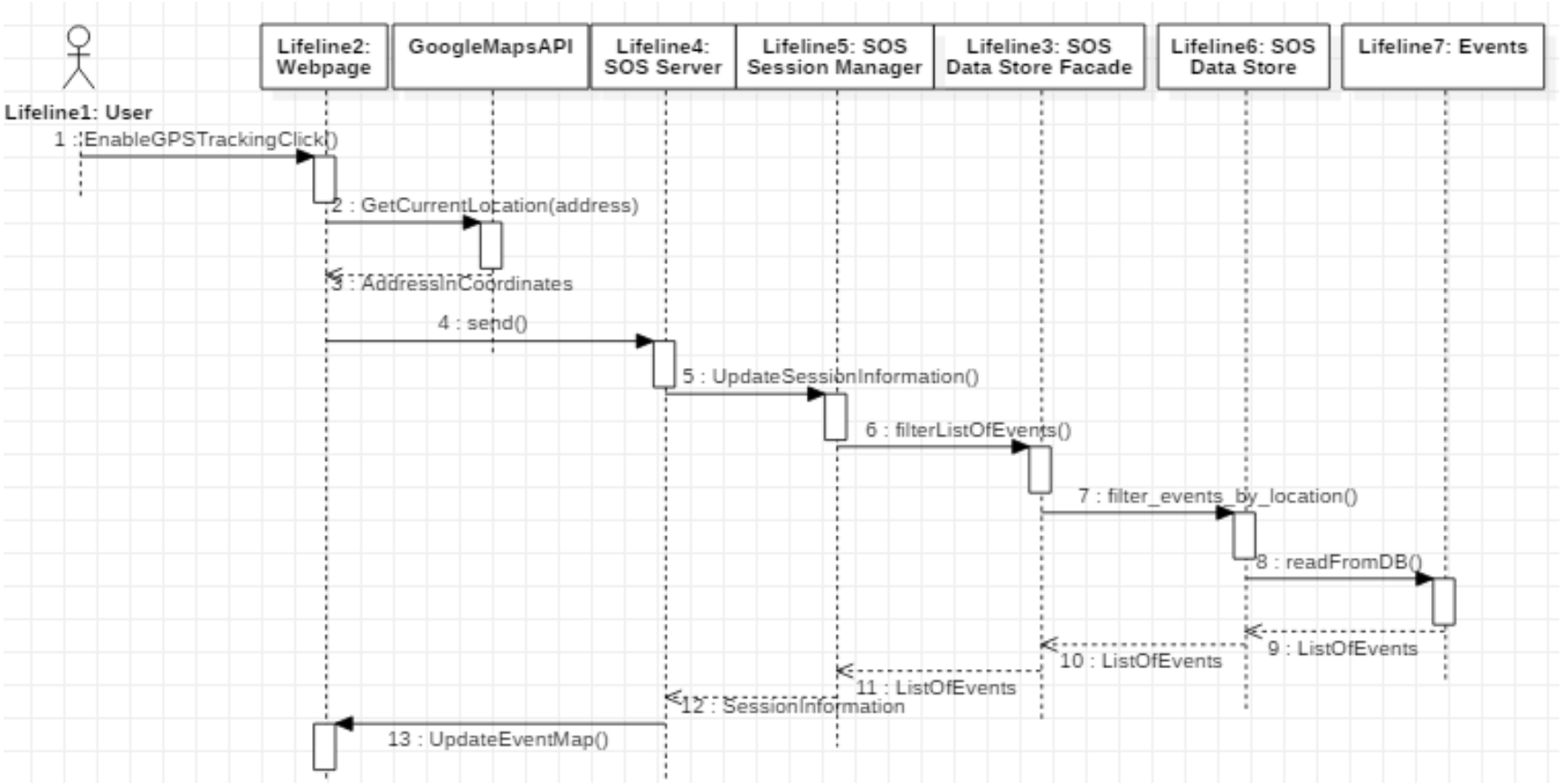

Figure 20: Sequence Diagram for SOS10 – Access an Event by Location.

3.3.9 Sequence Diagram for SOS31 – Login

The sequence diagram in Figure 21 corresponds to the Use Case in Appendix B, Section 7.2.9.

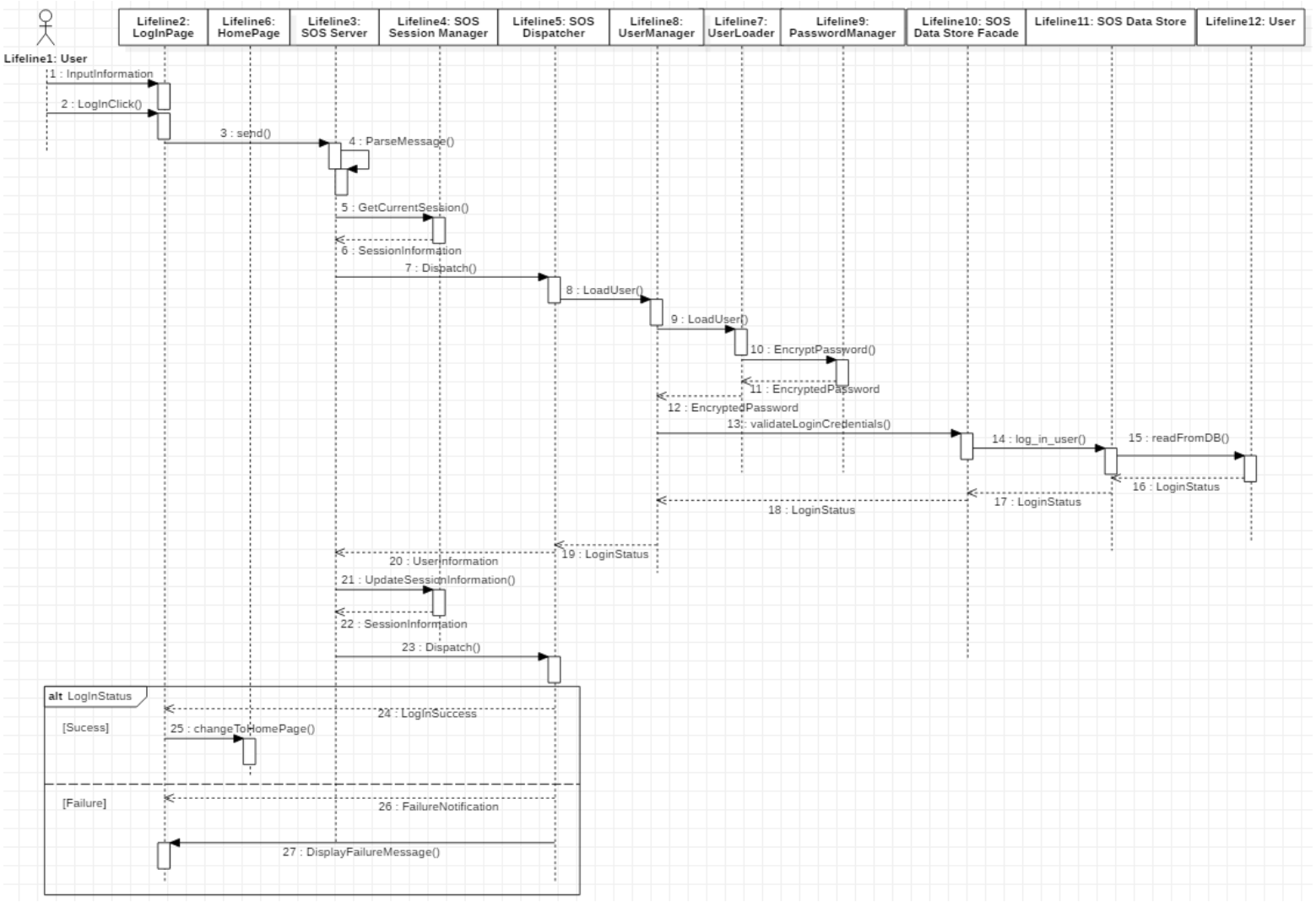

Figure 21: Sequence Diagram for SOS31 – Login.

3.3.10 Sequence Diagram for SOS32 – Logout

The sequence diagram in Figure 22 corresponds to the Use Case in Appendix B, Section 7.2.10.

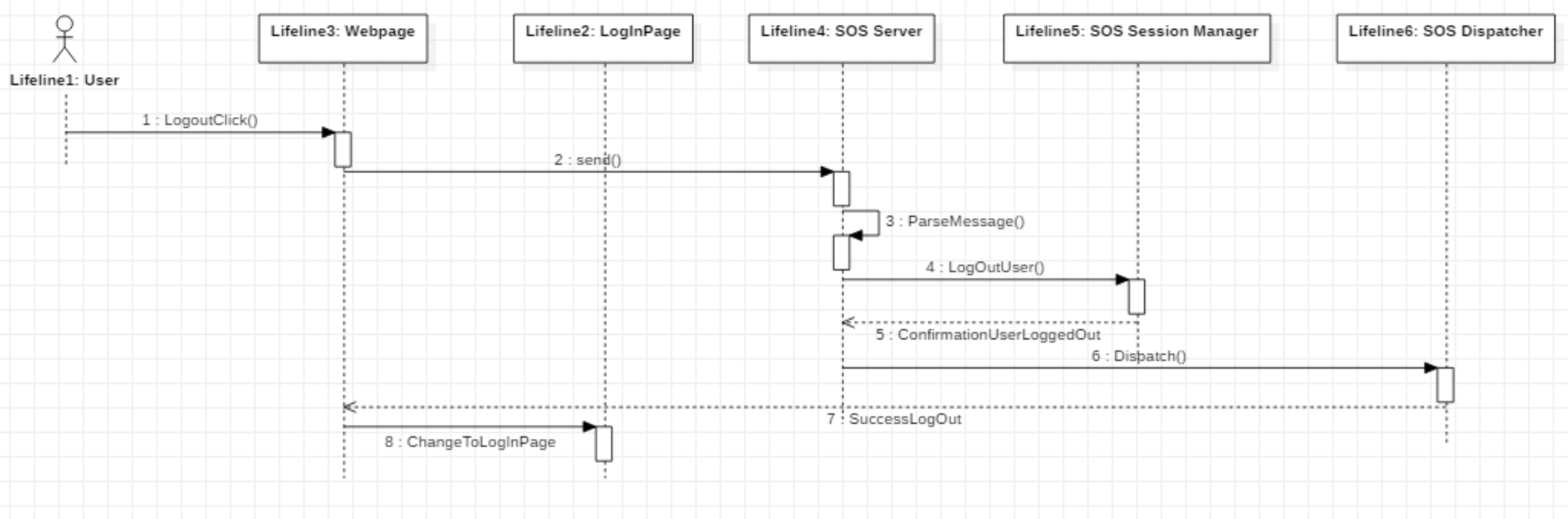

Figure 22: Sequence Diagram for SOS32 – Logout

3.4 Detailed Class Design

This section contains the detailed class design of each class in each subsystem of the SOS system. Section 3.4.1 contains a detailed description for each class in the system, while Section 3.4.2 contains the Object Constraint Language (OCL) constraints for each control objects of each subsection.

3.4.1 Class Description

Each one of the following subsections contains a detailed class description for each of the classes in the subsystems of the SOS. In each case, a minimal class diagram can be found in Section 3.1, and a complete class diagram can be found in Appendix C.

3.4.1.1 SOS Website

The complete class diagram for the SOS Website, which contains operations and attributes where applicable, can be found in Figure 30 in Appendix C, Section 7.3.1. The Java Code interface for these classes are in Appendix D, Section 7.4.1. The following classes are part of the SOS Website:

- Webpage, which is the core website class of the system, and the only one that interacts with the backend. Most other classes inherit from this one, namely all the ones with "Page" in their name. This means that all pages inherit the functionalities form the Webpage, including simple web-app functions like refresh, and more complex ones such as interfacing with the Google Maps GPS API.
- Session, which is a data wrapper class which contains client-specific information about the current session, such as what user is logged in, what their privileges are, current events, and so on. Most of the page information will be stored in this class and accessed rather than directly in pages, and all pages have access to it as they inherit it from Webpage.
- ProfilePage, which contains the User profile. This page presents a view of the User which is different depending on whether the User is seeing their own page, or someone else's page. If a User is logged in an seeing their own page, they have access to editing their profile and changing their privacy settings.
- EventPage, which contains the Event data. This page presents a view of an Event created by an Organization. It should also provide the attendance functionality, but only if the set time for the event and the current time of the system are the same. Event owners (i.e., organizers) also have access to canceling and editing event details.
- EventsPage, which contains a set or list of Events. This class represents a groping of Event classes, each of which can be accessed independently. On top of wrapping a list of events, this class also provides other functionalities such as filtering (namely by GPS location).
- OrganizationPage, which contains the Organization Data. This page presents a view of the Organization which is different depending on whether the User who is seeing it has privileges on the Organization (i.e., is an Organizer). For non-privilege Members, the Organization page just displays relevant information such as the about and description, but for privilege Organizers, it allows Event Creation, Event Cancellation, Member Invites, etc.

- OrganizationsPage, which contains a set or list of Organizations. This class represents a grouping of Organization classes, each of which can be accessed independently. On top of wrapping a list of organizations, this class also provides other functionalities such as filtering, and creating new organizations.
- HomePage, which contains the home page of the system. This is the first view of the system that users see and provides them with navigation links to all other pages.
- Form, which is the parent class for a series of input forms which are used throughout the other pages. As such, it includes several general functions which are inherited by other forms and that implement important features like submitting. Forms also have their own attributes as they don't use the same Session object as the pages.
- LoginForm, which is the form for user login. It stores the username and password and communicates them to the SOS Interface. It resides in the login page.
- RegistrationForm, which is the form for user registration. It stores username, password, confirm password and email and communicates it to the SOS Interface. It resides in the LoginPage.
- CreateEventForm, which is the form for creating an Event. It stores an event name, date, time, duration, type, visibility, and location and communicates it to the SOS Interface. It resides in the EventPage.
- EditProfileForm, which is the form for editing a User profile. It stores an email, privacy setting, date of birth, phone number, and password. Note that when open, these values are preset form the known current value of these attributes in the datastore. It resides inside the ProfilePage.
- CreateOrganizationForm, which is the form for creating a new Organization. It stores an organization name, description, privacy setting, and join requirement and communicates it to the SOS Interface. It resides inside the OrganizationPage.
- RoleCreationForm, which is the form for creating a new Role. It stores a role name, privileges, and security requirements and communicates those to the SOS Interface. It resides inside the OrganizationPage.

3.4.1.2 SOS Interface

The complete class diagram for the SOS Interface, which contains operations and attributes where applicable, can be found in *Figure 31* in Appendix C, Section 7.3.2. The Java Code interface for these classes are in Appendix D, Section 7.4.2. The following classes are part of the SOS Interface:

- SOS Server, which is the main server instance of the system. It implements the socket connections which receive network messages (Requests) from the remote front-end and preprocess them to prepare a dispatched request through SOS Dispatcher. The SOS Server also contains an instance of the SOS Session Manager class to keep and return Session data to the User when necessary. Note that this class is a Singleton, as only one SOS Server is necessary for the entire system.
- SOS Session Manager, which has functions relating to system session. It oversees controlling real-time elements of the system (e.g., system time, which affects the Events that Users can Attend, as well as past and future Events) and keeps track of logged-in Users

and their local actions when applicable. It is a Singleton, as only one Session Manager is necessary and more than one could create session conflicts that must be avoided.

- SOS Dispatcher, which is a Command class which propagates the front-end requests to their specific target controllers. The requests messages which are parsed and pre-processed by the SOS Server then are used to call an appropriate dispatch from the SOS Dispatcher, which is tasked with calling all other controllers. Instead of being their own classes, each subcommand is defined in terms of parametrizations of the dispatch function within the SOS Dispatcher. Internally, SOS Dispatcher also keeps track of these requests and stores them in the database.

3.4.1.3 User Management

The complete class diagram for the User Management, which contains operations and attributes where applicable, can be found in Figure 32 in Appendix C, Section 7.3.3. The Java Code interface for these classes are in Appendix D, Section 7.4.3. The following classes are part of the User Management:

- UserManager, which is a Singleton class which managers all the User functions. This class receives dispatched actions from the SOS Dispatcher and completes that action using objects internal to its subsystem. It also is in charge of interacting with the SOS Data Store Façade directly. Part of the role of this class is to parse front-end format user data (e.g., JSON-String defining a new User) and calling the appropriate functions on the other classes according to that data. It also tasked with encoding a User object into database format objects (e.g., SQL Table entry for User).
- NewUserBuilder, which is a Builder which creates new User objects. It is used to decouple the parts of the process of creating a new User from the actual User class, which is intended to only be a data wrapper class which can be easily parsed into the database format. Namely, this class implements the checks and validations necessary to create a valid User and will reject invalid ones. As part of this validation, it must interact with the SOS Security System classes that implement the password and access policies.
- UserLoader, which is a class which creates a User object from a database-format User object (e.g., a SQL Table entry for User). This class decouples the parsing from the database to the system logic from the UserManager class and can be extended to include internal checks for data integrity purposes.
- UserUpdater, which is a class which deals with User modifications. User modifications are done on the system logic-level User object first and are only finalized once they are stored to the database. The UserUpdater decouples these modifications from the UserManager class and from the User class itself and implements checks and validations in the same way that NewUserBuilder does. It also ensures that every modification to the User class is saved to the SOS Data Store.
- User, which is a run-time representation of a User persistent object. This class is used as an intermediary for creation, retrieval, and modification of User data within the Java code (and the JVM). It is encodable (or serializable) to a database format (e.g., SQL Entry).

3.4.1.4 Event Management

The complete class diagram for the Event Management, which contains operations and attributes where applicable, can be found in Figure 33 in Appendix C, Section 7.3.4. The Java Code interface for these classes are in Appendix D, Section 7.4.4. The following classes are part of the Event Management:

- EventManager, which is a Singleton which manages all the Event functions. This class receives dispatched actions from the SOS Dispatcher and completes that action using objects internal to its subsystem. It also is in charge of interacting with the SOS Data Store Façade directly. Part of the role of this class is to parse front-end format data (e.g., JSON-String description of new Events) and calling the appropriate functions on other classes according to that data. It is also in charge of encoding Event objects into database-format (e.g., SQL Table entries). Another role is to create EventLists based on filter requests through the EventListBuilder.
- EventBuilder, which is a Builder class which creates new Events. It is used to decouple the parts of the process of creating a new Event from the actual Event class, which is intended to only be a data wrapper class which can be easily parsed into the database format. Namely, this class implements the checks and validations necessary to create a valid Event and will reject invalid ones.
- EventLoader, which is a class which creates an Event object from an Event database object. This class decouples the parsing from the database to the system logic from the EventManager class and can be extended to include internal checks for data integrity purposes.
- EventListBuilder, which is a Builder which creates new EventList objects. As other builders, it is used to decouple the process of creating a new EventList from the actual EventList class, and also provides functions implementing attribute-base filtering (e.g., filter by location, or by hosting organization, etc.)
- EventList, which is a class that aggregates Events.
- Event, which is a run-time representation of an Event persistent Object. This class is used as an intermediary for creation, retrieval, and modification of Event data within the Java code (and the JVM). It is encodable (or serializable) to a database format (e.g., SQL Entry).

3.4.1.5 Organization Management

The complete class diagram for the Organization Management, which contains operations and attributes where applicable, can be found in Figure 34 in Appendix C, Section 7.3.5. The Java Code interface for these classes are in Appendix D, Section 7.4.5. The following classes are part of the Organization Management:

- OrganizationManager, which is a Singleton which manages all the Organization functions. This class receives dispatched actions from the SOS Dispatcher and completes that action using objects internal to its subsystem. It also is in charge of interacting with the SOS Data Store Façade directly. Part of the role of this class is to parse front-end format data (e.g., JSON-String description of new Organization) and calling the appropriate functions on other classes according to that data. Another job of this class is to manage Role creation

and assignment, as well as mediate the modification of data in an Organization, and of Event hosting.

- OrganizationBuilder, which is a Builder which creates new Organization objects. It is used to decouple the process, including validations and checks, of creating an Organization from the actual Organization class itself.
- OrganizationLoader, which is a class which creates an Organization object from an Organization database object. This class decouples the parsing from the database to the system logic from the OrganizationManager class and can be extended to include internal checks for data integrity purposes.
- Organization, which is a run-time representation of an Organization persistent object. This class is used as an intermediary for creation, retrieval, and modification of Organization data within the Java code (and the JVM). It is encodable (or serializable) to a database format (e.g., SQL Entry).

3.4.1.6 Security Management

The complete class diagram for the Security Management, which contains operations and attributes where applicable, can be found in Figure 35 in Appendix C, Section 7.3.6. The Java Code interface for these classes are in Appendix D, Section 7.4.6. The following classes are part of the Security Management:

- PasswordManager, which is a Singleton which deals with password control actions. It implements most of the back-end side of the password policy defined in Section 2.5.1, including resolving passwords and checking the input password against the database.
- AccessManager, which is a Singleton dealing with access control actions. It implements most of the back-end side of the access policy defined in Section 2.5.2 and host the relevant Enumerations for access permissions and other privileges. It also must be called to do checks on the relevant actions, such as creating events, deleting profiles, etc.

3.4.1.7 SOS Storage

The complete class diagram for the SOS Storage, which contains operations and attributes where applicable, can be found in Figure 36 in Appendix C, Section 7.3.7. The Java code interface for these classes is in Appendix D, Section 7.4.7. The following classes are part of the SOS Storage:

- SOS Data Store Façade, which is a Façade object that acts as the interface for the SOS Storage subsystem. The other subsystems interact with the database through a preset set of actions defined in the SOS Data Store Façade. A façade is a structural design pattern which is used to provide a unified interface to a set of objects within a subsystem. Even though our data store is a single object, a façade is still warranted because the SOS Data Store is implemented using a database component (SQL) and through the SOS Data Store Façade we can decouple the details of the database component (such as the SQL language) from the rest of the system.
- SOS Data Store, which is the actual database implementation for the SOS Storage Subsystem. The other subsystems interact with it through the SOS Data Store Façade. This class implements the system data storage and is effectively the repository (or repository

interface) in the Repository architecture of our system. The database component it links to is a relational (SQL) database file which implements the data management policy described in Section 2.4.

- User, which represents the User table on the database.
- Organizations, which represents the Organizations table on the database.
- Roles, which represents the Roles table on the database.
- Attendance, which represents the Attendance table on the database.

3.4.2 Control Objects Description

This section contains the Object Constraint Language (OCL) specification of the interfaces of each of the major control objects in the SOS subsystems. The OCL defines interfaces in terms of three types of constraints on classes: (a) an invariant, which is a predicate that is always true for a class; (b) a precondition, which is a predicate that must be true before the class is used; and (c) a postcondition, which is a predicate that must be true after the class is used.

3.4.2.1 SOS Server OCL

The OCL specification for the SOS Server class is defined in Figure 23.

```
context SOS Server
inv: self.sessionManager <> null

context SOS Server :: ParseMessage(JSONString)
pre:    json.ofCorrectFormat(JSONString)
        and JSONString.contains(action)
        and JSONString.contains(payload)

context SOS Server :: send (action, payload)
pre:    SOS Dispatcher.acceptedActions.includes(action)
        and action.ofCorrectFormat(payload)
```

Figure 23: OCL specification for SOS Sever.

3.4.2.2 User Manager OCL

The OCL specification for the User Manager class is defined in Figure 24.

```
context User Manager :: LoadUser (SQLEntry)
pre:    SQLEntry.contains(name)
        and SQLEntry.contains(user_name)
        and SQLEntry.contains(email)
        and SQLEntry.contains(password)
        and SQLEntry.contains(user_id)
        and SQLEntry.contains(privacy)

context User Manager :: changeUserDetails (User, change)
pre:    change.contains(name)
        or change.contains(user_name)
        or change.contains(email)
        or change.contains(privacy)
post:   User.name == change.name
        or User.user_name == change.user_name
        or User.email == change.email
        or User.privacy == change.privacy

context User Manager :: CreateNewProfile (JSONString)
pre:    JSONString.contains(name)
        and JSONString.contains(user_name)
        and JSONString.contains(email)
        and JSONString.contains(password)
```

Figure 24: OCL specification for User Manager.

3.4.2.3 Organization Manager OCL

The OCL specification for the Organization Manager class is defined in Figure 25: OCL

```
context Organization Manager :: grantRole(user_id, organization_id, privilege_ids)
pre:    SOS Database.User.contains(user_id)
        and privilege_ids->any(SOS Database.Privileges.contains(self))
post:   SOS Database.Roles.contains((user_id, organization_id))
```

specification for Organization Manager..

Figure 25: OCL specification for Organization Manager.

3.4.2.4 Event Manager OCL

The OCL specification for the Event Manager class is defined in Figure 26.

```
context Event Manager  :: cancelEvent (event_id)
pre:    SOS Database.Event.contains(event_id)
        and SOS Database.Event.get(event_id).cancelled == False
post:   SOS Database.Event.get(event_id).cancelled == True

context Event Manager  :: createEvent (JSONString)
pre:    JSONString.contains(location)
        and JSONString.contains(description)
        and JSONString.contains(date)
        and JSONString.contains(time)
        and JSONString.contains(event_type)
        and JSONString.contains(hosted_by)
        and JSONString.contains(visibility)

context Event Manager :: retrieveListOfEvents (event_ids)
pre:    event_ids.size() > 0 and event_ids->any(SOS Database.Event.contains(self))

context Event Manager :: markAttendance (user_id, event_id)
pre:    SOS Database.Event.contains(event_id)
        and SOS Database.User.contains(user_id)
post:   SOS Database.Attendance.contains((user_id, event_id))

context Event Manager :: getEventDetails (event_id)
pre:    SOS Database.Event.contains(event_id)
```

Figure 26: OCL specification for Event Manager.

3.4.2.5 PasswordManager OCL

The OCL specification for the PasswordManager class is defined in Figure 27.

```
context PasswordManager
inv:    self.passwordPolicy <> null
```

Figure 27: OCL specification for PasswordManager.

3.4.2.6 AccessManager OCL

The OCL specification for the AccessManager class is defined in Figure 28.

```
context PasswordManager
inv:    self.accessPolicy <> null
```

Figure 28: OCL specification for AccessManager.

4 Glossary

- **Scenario**, a scene that illustrates some interactions of the proposed system.
- **Static Model**, a model which does not depend on elements of time.
- **Dynamic Model**, a model which depends on or contains elements of time, especially allowing interactions between entities over time.
- **Gantt Chart,** a bar chart where the x-axis is time and the y-axis is the different tasks, and the duration of each task is represented by the length of a bar.
- **Unified Software Development Model**, …
- **Sequence Diagram,** an interaction diagram which focus on the time-ordering of messages and interactions.
- **Use Case Diagram,** a diagram that shows a set of use cases and actors; and their relations.
- **SOS,** Student Organization System.
- **Object Diagram,** a diagram that models the instances of things contained in a class diagram, i.e., a set of objects and their relationships at a point in time.
- **Class Diagram,** a UML diagram containing a representation
- **Attribute,** a variable on a UML class.
- **Operation,** a function on a UML class indicating an action.
- **Role,** a set of technical and managerial tasks that are expected from a participant or a team.
- **Activity,** a set of tasks performed towards a specific purpose.
- **Task,** an atomic unit of work that can be managed and that consumes resources.
- **Milestone,** end-point of a software process activity.
- **Deliverable,** a work product for the client.
- **Notation,** a graphical or textual set of rules representing a model.
- **Method,** a repeatable technique for solving a specific problem.
- **Methodology,** a collection of methods for solving a class of problems.
- **Use Case,** a sequence of events describing all possible actions between actors and the system for a given piece of functionality.
- **Actors,** the roles interacting with the system such as end-users and other computer systems.

5 Approval Page:

Approval Page of System Requirements Document of

Student Organization System

Member Signatures

Armando J. Ochoa	11/12/2019
Member Signature	Date
Yovanni Jones	11/12/2019
Member Signature	Date
M.Kian Maroofi	11/12/2019
Member Signature	Date
Teriq Douglas	11/12/2019
Member Signature	Date
Anthony Sanchez-Ayra	11/12/2019
Member Signature	Date

6 References

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

7 Appendices

7.1 Appendix A – Use Case Diagram

The Use Case Diagram for the Student Organization System is contained in Figure 29.

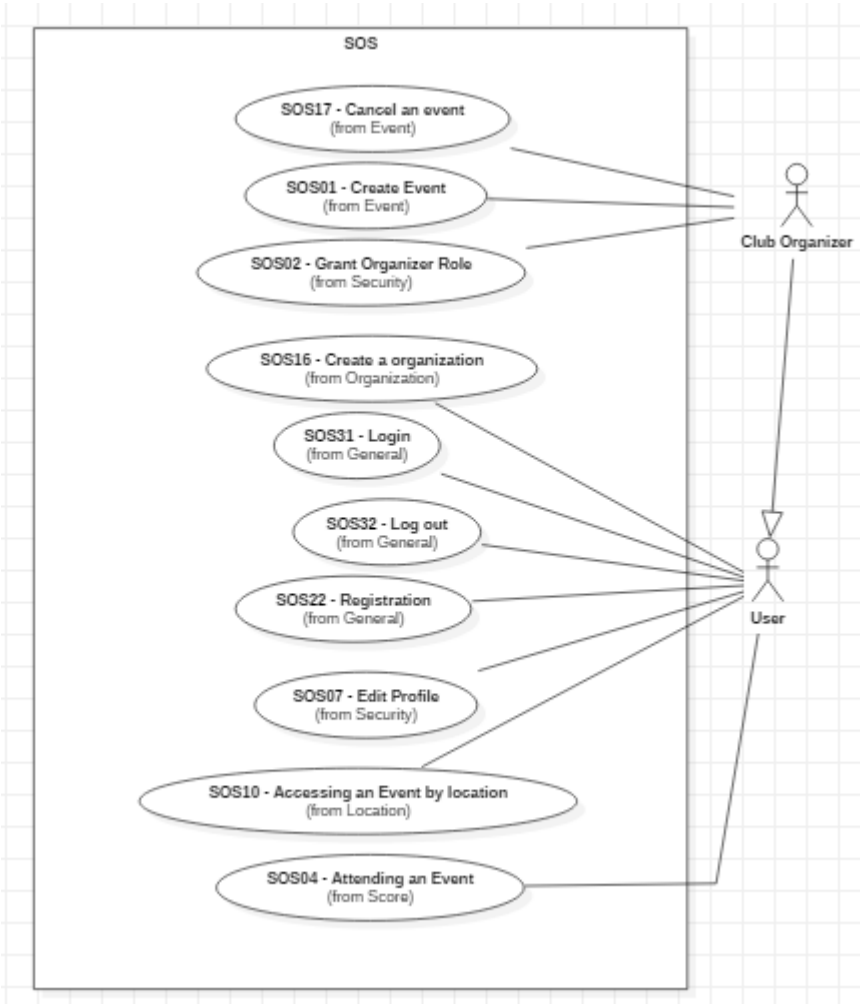

Figure 29: Use case diagram for the implemented Use Cases.

7.2 Appendix B – Implemented Use Cases

7.2.1 Create Event

Use Case ID: SOS01 - Create Event

Use Case Level: User Goal

Details:

- **Actor:** Organizer
- **Pre-conditions:**
    1. Organizer has successfully logged onto the system.
    2. Organizer is assigned to an Organization.
    3. Organizer has Event Creation privileges
- **Description:**
    1. Use case begins when Organizer clicks on **Create Event** on the administration page of their organization.
    2. The system shall prompt the Organizer with an Event Creation form, which shall present them with a template for data entry.
    3. The Organizer shall enter the following data:
        - **Event Name**
        - **Event Date and Time**
        - **Event Location**
        - **Event Description** (Optional)
        - **Event Type** (Defaults to Normal Event)
        - **Event Visibility** (Defaults to Visible)
    4. The Organizer shall complete the Event Creation by selecting the **publish** button.
    5. The system shall notify the Organizer that the event was published correctly.
    6. Use case ends when the system receives the Event specifications, generates a **unique event id** and publishes the Event according to the given specifications.
- **Relevant requirements:**

    None
- **Post-conditions:**
    1. An event has been published by the Organizer representing the Organization according to the specifications given.

Alternative Courses of Action

1. In step D.4, the Organizer has the option to **cancel** the Event Creation.
2. In step D.4, the Organizer has the option to **schedule** the Event Creation for a future date.
3. In step D.4, the Organizer has the option to **save without publishing** the Event Creation to complete at a later date.
4. In step D.5, if any of the required fields are blank, the system shall notify the Organizer and request an entry to the appropriate fields.

Extensions:

1. SOS21 – Avoid Time Conflicting Events

Exceptions:

1. The event database is not active.
2. The event creation view is not active.

Concurrent Uses:

None

Related Use Cases:

None

Decision Support

Frequency: On average 3 Events are created per Organization weekly.

Criticality: High. The most basic and central activity of the whole system is Event Creation.

Risk: Medium. Implementation does not require any complex specialized knowledge.

Constraints:

- Usability
    a) No previous training or knowledge.
    b) Tutorial or Help frame should be provided.
    c) Organizer should take less than 10 minutes to create an event.
- Reliability
    a) Mean Time to Failure – 5% failure monthly is acceptable.
    b) Availability
        - Downtime for Login Back-up – 30 minutes in a 24-hour period.
        - Downtime for Maintenance – 1 hour in a 2 weeks period.

- Performance
    a) The form should be sent and saved within 10 seconds.
    b) The system should be able to handle 50 requests in 1 minute.
- Supportability
    a) The Event Creation should be supported by Chrome, Mozilla, and IE.
- Implementation
    a) The implementation shall use JS React for front-end, and Java-based software for back-end.

Modification History

Owner: Armando J. Ochoa

Initiation date: 09/01/2019

Date last modified: 09/15/2019

7.2.2 Grant Organizer Role

Use Case ID: SOS2 – Grant Organizer Role

Use Case Level: User Goal

Details:

- **Actor:** Organizer
- **Pre-conditions:**
    1. Target Member belongs to the current organization.
    2. Target Member does not have Organizer status on the current organization.
    3. Organizer has power to give other people Organizer status.
- **Description:**
    1. Use case begins when the Organizer clicks on the **Add Organizer** tab on the organization management view.
    2. The system shall prompt the Organizer with an **Invitation Menu**, which shall present them with a template for data entry.
    3. The Organizer shall enter the following data:
        - **Member ID** (Either a name, or selectable from a drop-down menu with the list of organization members).
        - **Organizer Title** (Optional)
        - **Powers and Privileges** (From a list of pre-set privileges).
    4. The Organizer shall finish adding an organizer by selecting the **complete** button.

5. The system shall notify the Organizer that the Member's privilege and status has been changed correctly.
6. Use case ends when the system changes the Member's status in its database and the Member has been notified.

- **Post-conditions:**
  1. The status of the target Member has been changed, and he or she has received new privileges on the given organization.
  2. The list of Organizers in the Organization has been updated.
  3. The Member has been notified of the update.

Alternative Courses of Action

1. In step D.3, if the Organizer attempts to set a privilege that they themselves do not have, then the system shall notify them that they lack the required privileges (e.g., an Organizer without Event Creation privileges cannot invite another Organizer with Event Creation privileges).
2. In step D.4, the Organizer has the option to **cancel** the invitation.
3. In step D.5, if any of the required fields are blank, the system shall notify the Organizer and request an entry to the appropriate fields.

Extensions:

None

Exceptions:

1. Incorrect input in step D.3 (such as a non-existent Member ID) shall cause an exception and trigger a notification to the Organizer.

Concurrent Uses:

None

Related Use Cases:

None

Decision Support

Frequency: On average, 2 or 3 times per month per organization.

Criticality: High. This is basic element of the system and is required for good usability.

Risk: Medium. Implementation does not require any complex specialized knowledge.

Constraints:

- Usability
    a) No previous training or knowledge.
    b) Tutorial or Help frame should be provided.
    c) Organizer should take less than 10 minutes to complete the invitation.
- Reliability
    a) Mean Time to Failure – 1% failure yearly is acceptable.
    b) Availability – 30 minutes in a 24-hour period for backup and maintenance.
- Performance
    a) Privilege Checks should be done within 2 seconds.
    b) The system should handle 20 privilege checks in 1 minute.
- Supportability
    a) Should be supported by all browsers.
- Implementation
    a) Using Java-based software for back-end.

Modification History

Owner: Armando J. Ochoa

Initiation date: 09/01/2019

Date last modified: 09/15/2019

7.2.3 Attending an Event

Use Case ID: SOS04 - Attending an Event

Use Case Level: User Goal

Details:

- **Actor:** Member
- **Pre-conditions:**
    1. Member has an account in our application.
    2. Member is successfully logged into the application.
    3. Member is part of an organization and is attending an event hosted by said organization.
    4. Member is in the Events page and the relevant Events are loaded onto the page.
- **Description:**

    Trigger:

1. Use case begins when the Member click on the Event that they are currently attending.
2. The system shall provide the member with a description of the event as well as a button that says, "I'm here!"
3. The user shall click on the "I'm here" button.
4. The system shall process the request for the click.
5. Use case ends when the system notifies the user that their attendance at the event was noted.

- **Relevant requirements:**

  None

- **Post-conditions:**

1. The attendance request is saved in the system, along with arrival time.
2. The member is awarded a certain amount of points for attending the event.

Alternative Courses of Action:

1. In step D.10 the "I'm here" button will only appear if the user is at the location where the event is occurring.
2. In step D.8 the sorted list provided by to the user can be sorted by date the event will take place on or by organization name.

Exceptions:

1. If the member tries to click the I'm here button 15 minutes before the event is ending, they will not get credit for attending the event.

Concurrent Use Cases:

None.

Related Use Cases:

None.

Decision Support

Frequency: On average 100 attendance requests are made weekly by the organization leader.

Criticality: High. Allows the member to notify their organization that they are active in their organization.

Risk: High. Implementing this use case requires web-based technology and GPS tracking.

Constraints:

- Usability:
  a) No previous training required.

b) On average the user should take 2 minutes to complete the notification request to the system.

- Reliability
  a) Mean time to failure – 5% failures for every month of operation is acceptable.
  b) Availability – Down time for Login Back-up 30 minutes in a 24-hour period.
- Performance
  a) Request should be sent and saved within 10 seconds.
  b) System should be able to handle 20 requests in 1 minute.
- Supportability
  a) The Event Creation should be supported by Chrome, Mozilla, and IE.
- Implementation
  a) The implementation shall use JS React for front-end, and Java-based software for back-end.

Modification History

Owner: Anthony Sanchez-Ayra

Initiation date: 09/04/2019

Date last modified: 09/15/2019

7.2.4 Edit Profile

Use Case ID: SOS7 – Edit Profile

Use Case Level: Security

Details:

- **Actor:** User
- **Pre-conditions:**
  1. User have already signed up.
  2. User is currently at their profile page.
- **Description:**
  1. Use case begins when user clicks on the edit profile button.
  2. The system then will retrieve current user data by contacting the data storage and send the data back to the front-end.
  3. The page shall display the retrieved data in an input form which will allow the user to modify the data in the edit profile form:
     - Email
     - Phone number
     - Privacy

- Date Of Birth

4. The user inputs the modified data and clicks on the submit button.
5. The system shall ask the user for their password.
6. The user inputs their password and clicks confirm.
7. The system shall transmit the modified data to the data storage.
8. The case ends when there is a confirmation message.

- **Relevant requirements:**

  None.

- **Post-conditions:**

1. User information in the datastore has updated values.
2. Profile page has been updated with the updated values.

Alternative Courses of Action:

1. In step D.4, it is possible that the user closes the input form without clicking the submit button. In that case system shall not change the current user information.

Extensions:

None.

Exceptions:

None.

Concurrent Uses:

None

Related Use Cases:

SOS6 – Ensure User Profile Privacy

Decision Support

Frequency: On average, 20 Users will change their privacy settings on a given week.

Criticality: Low. User-set privacy is a secondary feature of the system.

Risk: Medium. This does not require any complex background knowledge except for some basic knowledge about access control.

Constraints:

- Usability
  a) No previous training or knowledge required to use this functionality.
  b) 1 Tutorial or Help frame should be provided.
  c) Users should take less than 10 minutes to find the functionality and correctly use it.

- Reliability
    a) Mean Time to Failure – 5% failure monthly is acceptable.
    b) Availability
        - Downtime for Login Back-up – 30 minutes in a 24-hour period.
        - Downtime for Maintenance – 1 hour in a 2 weeks period.
- Performance
    a) Privilege Checks should be done within 2 seconds.
    b) The system should handle 20 privilege checks in 1 minute.
- Supportability
    a) Should be supported by all browsers.
- Implementation
    a) Using Java-based software for back-end.

Modification History

Owner: Kian Maroofi

Initiation date: 09/10/2019

Date last modified: 09/27/2019

7.2.5 Access Events by Location

Use Case ID: SOS10 – Access Events by Location

Use Case Level: User Goal

Details:

- **Actor:** User
- **Pre-conditions:**
    1. User is logged into the system.
- **Description:**
    1. Use case begins when the User goes to the Events page or the Home page on the website.
    2. The webpage shall ask for accessing to the current location of the User by GPS.
    3. The system shall verify that User gave access to their location.
    4. The system shall find events within a defined proximity range of the User's location.
    5. The system shall update the Event map component to center on the User's location.
    6. The case ends when the system modifies the Event feed to prioritize Events within range of the User's location, and when the Event map component is updated to the User's location.

- **Relevant requirements:**

  None

- **Post-conditions:**

  1. The User's location is tracked on the system, and several Events are marked as within range.
  2. The Map component is updated to center on the User's location.

- **Alternative Courses of Action:**

  1. In step D.2, if the User has agreed to share location before, or if it has a permanent flag to share location in his or her profile, then it this step is ignored, and the system jumps directly to D.4
  2. In step D.3, if the User declines access, then the system shall ignore User location when presenting the Events.
  3. In step D.4, if location is not enabled, the system shall present all Events of the Organization.
  4. In step D.5, if location is not enabled, the system shall center on a system-wide default position.

Extensions:

None.

Exceptions:

None.

Concurrent Uses:

None

Related Use Cases:

None

Decision Support

Frequency: On average, users access the Home and Event pages 5 to 10 times daily.

Criticality: Medium, geolocation of events is an optional functionality that not everybody will use, and that is subordinate to other systems.

Risk: Medium. Medium. Implementation requires specialized knowledge, but GPS and Geolocation Services are available in most web browsers (Desktop and Mobile).

Constraints:

- Usability
  a) No previous training or knowledge required to use this functionality.
  b) 1 Tutorial or Help frame should be provided.
  c) Users should take less than 10 minutes to find the functionality and correctly use it.

- Reliability
    a) Mean Time to Failure – 1% failure yearly is acceptable.
    b) Availability – 30 minutes in a 24-hour period for backup and maintenance.
- Performance
    a) Privilege Checks should be done within 2 seconds.
    b) The system should handle 20 privilege checks in 1 minute.
- Supportability
    a) Should be supported by all browsers.
- Implementation
    a) Using Java-based software for back-end.

Modification History

Owner: Kian Maroofi

Initiation date: 09/10/2019

Date last modified: 09/15/2019

7.2.6 Create Organization

Use Case ID: SOS16 – Create Organization

Use Case Level: High-Level

Details:

- **Actor:** User
- **Pre-conditions:**
    1. User has an account in our application.
    2. User is successfully logged into the application.
    3. User is in the Organizations Page and the organizations they are part off are already loaded onto the screen.
- **Description:**
    1. Use case begins when User is on the Organizations page and they click on the "Create Organization" option.
    2. The User will click on the Create Organization option.
    3. The organization page shall provide the User with a form to fill out, asking for the following details:
        - **Organization Name**

- **Organization Description**
- **Requirements for Joining**
- **Privacy of the Organization** (whether it's open to others or not).

4. The User submits the club creation form.
5. The system shall notify the User that the request was submitted correctly by showing a notification in the Organization page.
6. Use case ends when the organization page the displays a notification that the User has created a new organization.

- **Relevant requirements:**

  None

- **Post-conditions:**

1. The request to create an organization is stored in the system.
2. The organization is shown to members depending on its privacy settings.
3. The User has gained owner status with respect to the created organization.

Alternative Courses of Action:

1. In step D.4 the user has the option to cancel the creation of their organization.
2. In step D.5 if any of the fields are left blank the system will provide the user with a message to fill in all the fields.
3. In step D.5 the system shall ask the user to confirm if they would like to create an organization.

Exceptions:

1. If the User tries to make an organization that already exists, then they will get an error message.

Concurrent Use Cases:

None.

Related Use Cases:

None.

Decision Support

Frequency: On average 20 organization creation requests are made monthly by the User.

Criticality: High. Allows the User to create an organization which allows new communities to grow around campus.

Risk: Medium. Implementing this use case requires web-based technology.

Constraints:

- Usability:
    a) No previous training required.
    b) On average the user should take 2 minutes to complete the notification request to the system.
- Reliability
    a) Mean time to failure – 5% failures for every month of operation is acceptable.
    b) Availability – Down time for Login Back-up 30 minutes in a 24 hour period.
- Performance
    a) Request should be sent and saved within 6 seconds.
    b) System should be able to handle 200 requests in 1 minute.
- Supportability
    a) The Event Creation should be supported by Chrome, Mozilla, and IE.
- Implementation
    a) The implementation shall use JS React for front-end, and Java-based software for back-end.

Modification History

Owner: Anthony Sanchez-Ayra

Initiation date: 09/04/2019

Date last modified: 09/15/2019

7.2.7 Cancel an Event

Use Case ID: SOS17 - Cancel an Event

Use Case Level: User Goal

Details:

- **Actor:** Organizer
- **Pre-conditions:**
    1. Organizer has an account in our application.
    2. Organizer is successfully logged into the application.
    3. Organizer is part of a organization.
    4. Organizer is on the Events page, where all events they have available is already loaded onto the page.
- **Description:**
    1. Use case begins when organizer clicks on the event that they want to cancel.

2. The system shall redirect the organizer to the Event Description view, which shall present them with a button labeled cancel event.
3. The organizer will click on the cancel event button.
4. The organizer will click yes on the validation message displayed by the system.
5. The system shall notify the organizer that the event was cancelled.
6. End case ends when the system removes the event from being viewed.

- **Relevant requirements:**

  None

- **Post-conditions:**

  1. The system notifies all users that subscribed to the event that it has been cancelled.

Alternative Courses of Action:

1. In step D.3 the system will prompt the organizer with a validation message to confirm that they actually want to cancel the event.

Exceptions:

1. The database is not active.
2. The Event Description view is not active.
3. The validation message is not active.

Concurrent Use Cases:

None.

Related Use Cases:

None.

Decision Support

Frequency: On average 5 cancellation requests are made weekly by the organizer.

Criticality: High. Allows the organizer to cancel an event whenever necessary.

Risk: High. Implementing this use case requires web-based technology.

Constraints:

- Usability:
  a) No previous training required.
  b) On average the user should take 2 minutes to complete the notification request to the system.
- Reliability

a) Mean time to failure – 5% failures for every month of operation is acceptable.

b) Availability – Down time for Login Back-up 30 minutes in a 24 hour period.

- Performance
    a) Request should be sent and saved within 6 seconds.
    b) System should be able to handle 10 requests in 1 minute.
- Supportability
    a) Shall should be supported by Chrome, Mozilla, and IE.
- Implementation
    a) The implementation shall use JS React for front-end, and Java-based software for back-end.

Modification History

Owner: Anthony Sanchez-Ayra

Initiation date: 09/04/2019

Date last modified: 09/15/2019

7.2.8 Registration

Use Case ID: SOS22 – Registration

Use Case Level: User Goal

Details:

- **Actor:** User
- **Pre-conditions:**
    1. The User does not have an account on the site.
- **Description:**
    1. Use case begins when the User presses the **Register** button on the log-in/register page.
    2. The system shall prompt the User with a **Registration** form, which shall present them with a template for data entry.
    3. The Organizer shall input the following data in the template:
        - **User Name**
        - **Email**
        - **Password**
        - **Confirm Password**
    4. The User shall complete the registration by selecting the **Ok** button.

5. The system shall confirm that the registration was successful.
6. Use case ends when the User is automatically logged into the system and the view is moved to home.

- **Relevant requirements:**

  None

- **Post-conditions:**

  None

Alternative Courses of Action:

1. In step D.3, If any of the fields have incorrect information or are left blank system will respond with a message saying that proper credentials should be entered.

Exceptions:

None

Concurrent Use Cases:

None.

Related Use Cases:

None.

Decision Support

Frequency: On average, 20 tasks are added to events a week.

Criticality: Medium. Not all events require tasks to be complete, so not all users will use this functionality.

Risk: Medium. Implementation does not require any complex specialized knowledge besides a database system.

Constraints:

- Usability:
  a) Requires minimal training.
  b) One or two help frames on the Help page shall be provided explaining how to add tasks.
  c) On average the user should less than 5 minutes to complete the notification request to the system.
- Reliability
  a) Mean time to failure – 5% failures for every 24 hours of operation is acceptable.
  b) Availability
     - Downtime for Login Back-up – 30 minutes in a 24-hour period.

- Downtime for Maintenance – 1 hour in a 2 weeks period.

- Performance
    a) Request should be sent and saved within 10 seconds.
    b) System should be able to handle 20 requests in 1 minute.
- Supportability
    a) Shall be supported by Chrome, Mozilla, and IE.
- Implementation
    a) The implementation shall use JS React for front-end, and Java-based software for back-end.

Modification History

Owner: Yovanni Jones

Initiation date: 09/02/2019

Date last modified: 09/22/2019

7.2.9 Log in

Use Case ID: SOS31 – Log in

Use Case Level: User Goal

Details:

- **Actor:** User
- **Pre-conditions:**
    1. The User has an account on the SOS site.
- **Description:**
    1. Use case begins when the user is in the **Log-In** page of the site.
    2. The login page shall provide an input form with to following parameters:
        - **Email address**
        - **Password**
    3. The user inputs their email and password and then clicks on login.
    4. The system shall verify if the email and password match.
    5. Use case ends when system allows the user to login.
- **Relevant requirements:**

    None
- **Post-conditions:**

1. the user is redirected to the **Home** page.

Alternative Courses of Action:

1. In step D.4, if the user types an invalid password or email then the system will notify them that their "email and password do not match."

Exceptions:

None

Concurrent Use Cases:

None.

Related Use Cases:

None.

Decision Support

Frequency: On average, up to 10000 requests daily.

Criticality: High. Allows the user to log-in to view their organizations and nearby events.

Risk: Low. Implementing this use case doesn't requires specified knowledge.

Constraints:

- Usability:
    a) Requires no training.
    b) On average the user should take less than 10 seconds to type their information and attempt to log in.
- Reliability
    a) Mean time to failure – 5% failures for every 24 hours of operation is acceptable.
    b) Availability
        - Downtime for Login Back-up – 30 minutes in a 24-hour period.
        - Downtime for Maintenance – 1 hour in a 2 weeks period.
- Performance
    a) Complete log-in should be done in at most 10 seconds.
- Supportability
    a) Should be supported by Chrome, Mozilla, and IE.
- Implementation
    a) The implementation shall use JS React for front-end, and Java-based software for back-end.

Modification History

Owner: Anthony Sanchez-Ayra

Initiation date: 09/06/2019

Date last modified: 09/16/2019

7.2.10 Log Out
Use Case ID: SOS32 – Log out

Use Case Level: User Goal

Details:

- **Actor:** User
- **Pre-conditions:**
    1. The User is currently logged into the SOS page.
- **Description:**
    1. Use case begins when the user clicks on the **Sign Out** button.
    2. The current page the user is in will call a system call to log the user out.
    3. The system will then attempt to log the user out of the webpage.
    4. Use case ends when website redirects the user to the **Login** page.
- **Relevant requirements:**

  None
- **Post-conditions:**

  None.

Alternative Courses of Action:

None.

Exceptions:

None.

Concurrent Use Cases:

None.

Related Use Cases:

None.

Decision Support

Frequency: On average, up to 10000 requests daily.

Criticality: High. Allows the user to log-out to make sure that no other user can tamper with their account if they were to access the site from the same computer.

Risk: Low. Implementing this use case doesn't requires specialized knowledge.

Constraints:

- Usability:
    a) Requires no training.
    b) On average the user should take less than 5 seconds to find the sign out button and click on it.
- Reliability
    a) Mean time to failure – 5% failures for every 24 hours of operation is acceptable.
    b) Availability
        - Downtime for Login Back-up – 30 minutes in a 24-hour period.
        - Downtime for Maintenance – 1 hour in a 2 weeks period.
- Performance
    a) Complete log-our should be done in at most 10 seconds.
- Supportability
    a) Should be supported by Chrome, Mozilla, and IE.
- Implementation
    a) The implementation shall use JS React for front-end, and Java-based software for back-end.

Modification History

Owner: Anthony Sanchez-Ayra

Initiation date: 09/06/2019

Date last modified: 09/16/2019

7.3 Appendix C – Detailed Subsystem Class Diagrams

7.3.1 SOS Website

The full class diagram can be seen in Figure 30.

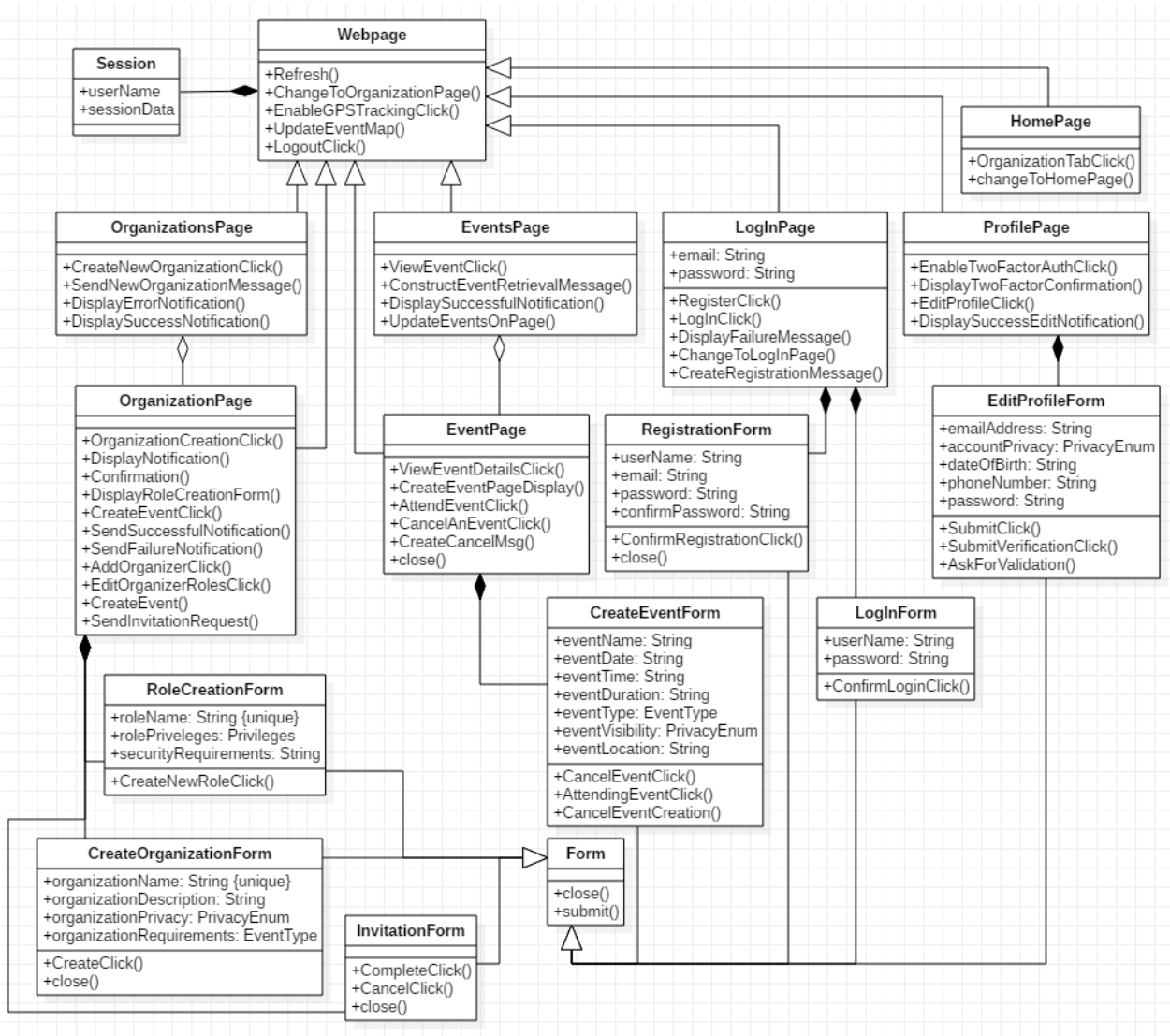

Figure 30: Full Class Diagram for the SOS Website subsystem.

7.3.2 SOS Interface

The full class diagram can be seen in Figure 31.

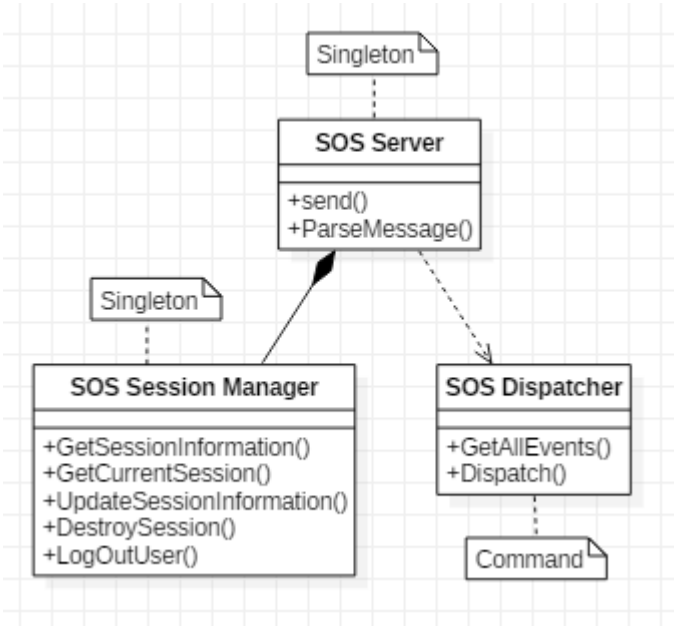

Figure 31: Full Class Diagram for the SOS Interface.

7.3.3 User Management

The full class diagram can be seen in Figure 32.

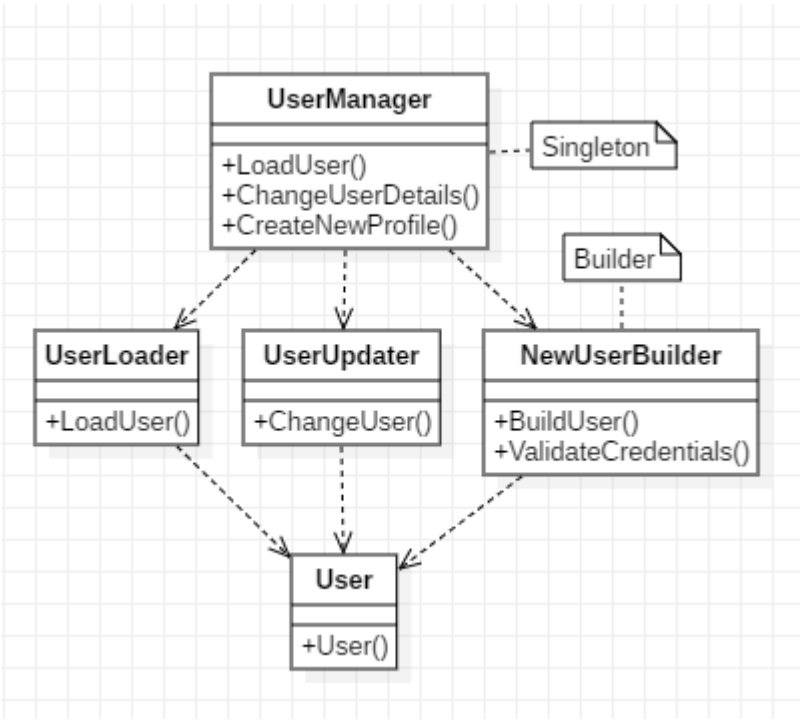

Figure 32: Full Class Diagram for the User Management.

7.3.4 Event Management

The full class diagram can be seen in Figure 33.

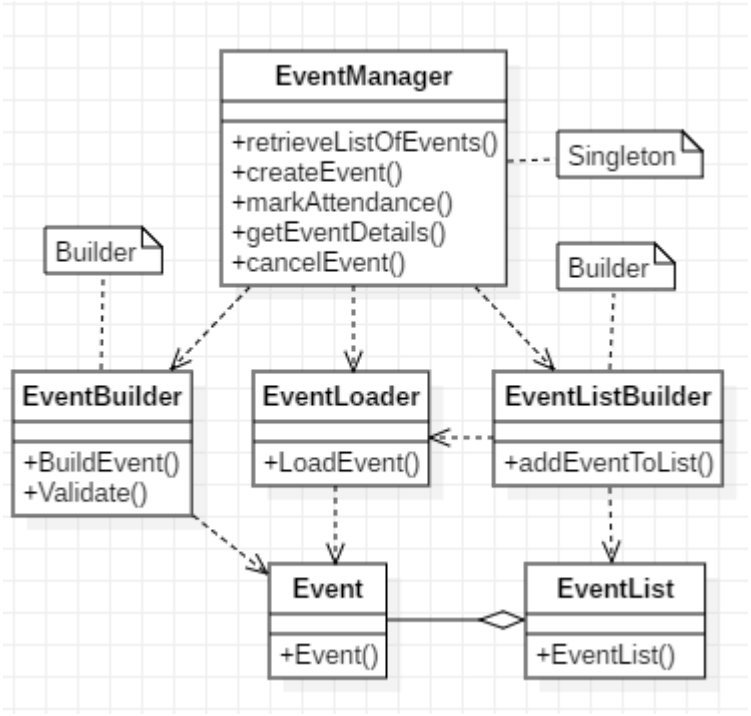

Figure 33: Full Class Diagram for the Event Management.

7.3.5 Organization Management

The full class diagram can be seen in Figure 34.

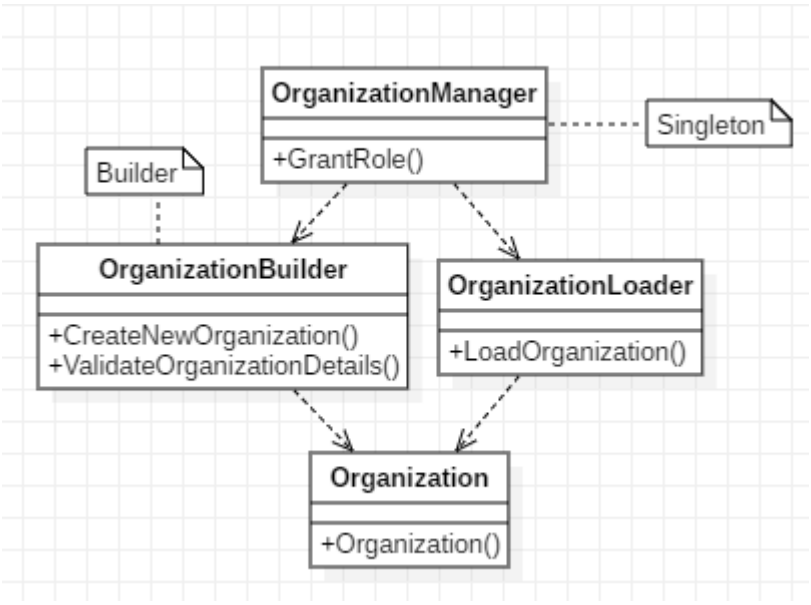

Figure 34: Full Class Diagram for the Organization System.

7.3.6 Security Management

The full class diagram can be seen in Figure 35.

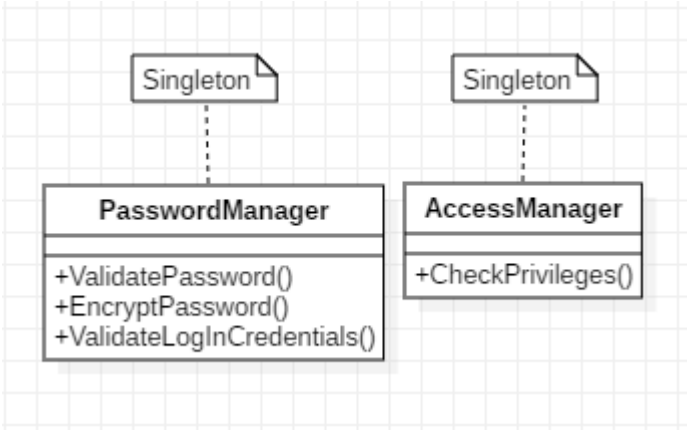

Figure 35: Full Class Diagram for the Security System.

7.3.7 SOS Storage

The full class diagram can be seen in Figure 36.

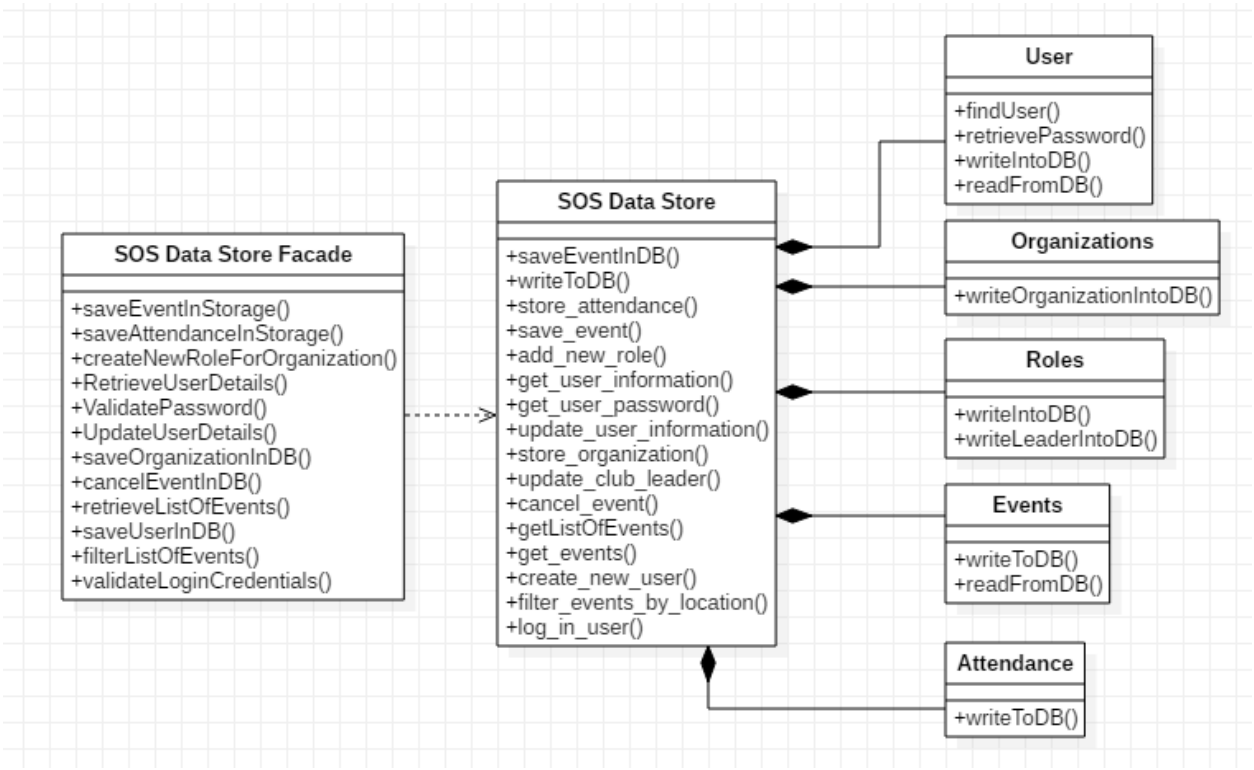

Figure 36: Full Class Diagram for the SOS Storage.

7.4 Appendix D - Class Interfaces

Package event

Class Summary

Event

A run-time representation of an Event persistent Object.

EventBuilder

A Builder class which creates new Events.

EventList

A class that aggregates Events.

EventListBuilder

A Builder which creates new EventList objects.

EventManager

EventManager, which is a Singleton which manages all the Event functions.

---

event

Class Event

java.lang.Object
|
+--event.Event

< Constructors >

---

public class **Event**
extends java.lang.Object

A run-time representation of an Event persistent Object. This class is used as an intermediary for creation, retrieval, and modification of Event data within the Java code (and the JVM). It is encodable (or serializable) to a database format (e.g., SQL Entry).

Constructors

Event

protected **Event**()

Constructs a new Event class. Called through the EventBuilder class. Attribute assignations are done through protected scope.

---

event

Class EventBuilder

java.lang.Object
|
+--event.EventBuilder

< Constructors > < Methods >

---

public class **EventBuilder**
extends java.lang.Object

A Builder class which creates new Events. It is used to decouple the parts of the process of creating a new Event from the actual Event class, which is intended to only be a data wrapper class. Namely, this class implements the checks and validations necessary to create a valid Event and will reject invalid ones.

Constructors

EventBuilder

public **EventBuilder**()

Creates a new EventBuilder to instantiate the new Event.

Methods

BuildEvent

public Event **BuildEvent**()

Finalizes the Event creation and returns it.

Returns:

returns the final Event.

Validate

protected boolean **Validate**()

Checks the current Event, returning True if it is valid so far and False otherwise.

Returns:

True, if the Event is valid so far. False if otherwise.

---

setDate

public EventBuilder **setDate**(java.lang.String date)

Adds a date value to the current Event.

Parameters:

date - the value to be added.

Returns:

the current builder.

---

setDescription

public EventBuilder **setDescription**(java.lang.String description)

Adds a description value to the current Event.

Parameters:

description - the value to be added.

Returns:

the current builder.

---

setEventtype

public EventBuilder **setEventtype**(java.lang.String eventType)

Adds a Event Type value to the current Event.

Parameters:

eventType - the value to be added.

Returns:

the current builder.

---

setHost

public EventBuilder **setHost**(java.lang.String host)

Adds a host value to the current Event.

Parameters:

host - the value to be added.

Returns:

the current builder.

---

setLocation

public EventBuilder **setLocation**(java.lang.String location)

Adds a location value to the current Event.

Parameters:

location - the value to be added.

Returns:

the current builder.

---

setName

public EventBuilder **setName**(java.lang.String name)

Adds a name value to the current Event.

Parameters:

name - the value to be added.

Returns:

the current builder.

---

setTime

public EventBuilder **setTime**(java.lang.String time)

Adds a time value to the current Event.

Parameters:

time - the value to be added.

Returns:

the current builder.

---

setVisibility

public EventBuilder **setVisibility**(java.lang.String visibility)

Adds a visibility value to the current Event.

Parameters:

visibility - the value to be added.

Returns:

the current builder.

---

event

Class EventList

java.lang.Object
|
+--event.EventList

< Constructors >

---

public class **EventList**
extends java.lang.Object

A class that aggregates Events.

Constructors

EventList

protected **EventList**()

Constructs a new EventList class. Called through the EventListBuilder class. Attribute assignations are done through protected scope.

---

event

Class EventListBuilder

java.lang.Object
|
+--event.EventListBuilder

< Constructors > < Methods >

public class **EventListBuilder**
extends java.lang.Object

A Builder which creates new EventList objects. As other builders, it is used to decouple the process of creating a new EventList from the actual EventList class, and also provides functions implementing attribute-base filtering (e.g., filter by location, or by hosting organization, etc.)

Constructors

EventListBuilder

public **EventListBuilder**()

Creates a new EventListBuilder to instantiate the new EventList.

Methods

setAddeventtolist

public EventListBuilder **setAddeventtolist**(java.lang.String addEventToList)

Adds an Event value to the current EventList.

Parameters:

addEventToList - the value to be added.

Returns:

the current builder.

---

event

Class EventManager

java.lang.Object
|
+--event.EventManager

< Constructors > < Methods >

public class **EventManager**
extends java.lang.Object

EventManager, which is a Singleton which manages all the Event functions. This class receives dispatched actions from the SOS Dispatcher and completes that action using objects internal to its subsystem. It also is in charge of interacting with the SOS Data Store Façade directly. Part of the role of this class is to parse front-end format data (e.g., JSON-String description of new Events) and calling the appropriate functions on other classes according to that data. It is also in charge of encoding Event objects into database-format (e.g., SQL Table entries). Another role is to create EventLists based on filter requests through the EventListBuilder.

Constructors

EventManager

protected **EventManager**()

A protected or private constructor ensures that no other class has access to the Singleton.

Methods

cancelEvent

public void **cancelEvent**(int event_id)

Sets the is_cancelled property of the given Event to True.

Parameters:

event_id - the wanted Event.

---

createEvent

public Event **createEvent**(java.lang.String jsonString)

Creates a new Event from a json Event description. Done by calling the EventBuilder class.

Parameters:

jsonString - the JSON object describing the new Event.

Returns:

a Event object with the given attributes.

---

getEventDetails

public Event **getEventDetails**(int event_id)

Loads an Event with the given Event id.

Parameters:

event_id - the id of the wanted Event.

Returns:

the Event with the corresponding id.

---

instance

public static EventManager **instance**()

Returns:

The unique instance of this class.

---

markAttendance

public void **markAttendance**(int user_id,
int event_id)

Marks a User as attending an Event by creating an entry on the Attendance table.

Parameters:

user_id - the id of the User event_id
- the id of teh Event

---

retrieveListOfEvents

public EventList **retrieveListOfEvents**(int[] event_ids)

Retrieves a list of Events in the for of an EventList. This is done through an EventListBuilder.

Parameters:

event_ids - the ids of the Events to be added.

Returns:

the EventList containing the given Events.

Package organization

Class Summary

Organization

A run-time representation of an Organization persistent object.

OrganizationBuilder

A Builder which creates new Organization objects.

OrganizationLoader

A class which creates an Organization object from an Organization database object.

OrganizationManager

A Singleton which manages all the Organization functions.

---

organization

Class Organization

java.lang.Object
|
+--organization.Organization

< Constructors >

---

public class **Organization**
extends java.lang.Object

A run-time representation of an Organization persistent object. This class is used as an intermediary for creation, retrieval, and modification of Organization data within the Java code (and the JVM). It is encodable (or serializable) to a database format (e.g., SQL Entry)

Constructors

Organization

protected **Organization**()

Constructs a new Organization class. Called through the OrganizationBuilder class. Attribute assignations are done through protected scope.

---

organization

Class OrganizationBuilder

java.lang.Object
|
+--organization.OrganizationBuilder

< Constructors > < Methods >

---

public class **OrganizationBuilder**
extends java.lang.Object

A Builder which creates new Organization objects. It is used to decouple the process, including validations and checks, of creating an Organization from the actual Organization class itself.

Constructors

OrganizationBuilder

public **OrganizationBuilder**()

Creates a new OrganizationBuilder to instantiate the new Event.

Methods

CreateNewOrganization

public Organization **CreateNewOrganization**()

Finalizes the Organization creation and returns it.

Returns:

returns the final Organization.

ValidateOrganizationDetails

protected boolean **ValidateOrganizationDetails**()

Checks the current Organization, returning True if it is valid so far and False otherwise.

Returns:

True, if the Organization is valid so far. False if otherwise.

---

setDescription

public OrganizationBuilder **setDescription**(java.lang.String description)

Adds a description value to the current Organization.

Parameters:

description - the value to be added.

Returns:

the current builder.

---

setName

public OrganizationBuilder **setName**(java.lang.String name)

Adds a name value to the current Organization.

Parameters:

name - the value to be added.

Returns:

the current builder.

---

setPrivacy

public OrganizationBuilder **setPrivacy**(java.lang.String privacy)

Adds a privacy value to the current Organization.

Parameters:

privacy - the value to be added.

Returns:

the current builder.

---

setRequirements

public OrganizationBuilder **setRequirements**(java.lang.String requirements)

Adds a requirements value to the current Organization.

Parameters:

requirements - the value to be added.

Returns:

the current builder.

---

organization

Class OrganizationLoader

java.lang.Object
|
+--organization.OrganizationLoader

< Constructors > < Methods >

---

public class **OrganizationLoader**
extends java.lang.Object

A class which creates an Organization object from an Organization database object. This class decouples the parsing from the database to the system logic from the OrganizationManager class and can be extended to include internal checks for data integrity purposes.

Constructors

OrganizationLoader

public **OrganizationLoader**()

Methods

LoadOrganization

public static Organization **LoadOrganization**(java.lang.String sqlEntry)

Creates a Organization from a database-format entry.

Parameters:

sqlEntry - a sql entry for the given organization.

Returns:

a Organization object with the given attributes.

---

organization

Class OrganizationManager

java.lang.Object
|
+--organization.OrganizationManager

< Constructors > < Methods >

---

public class **OrganizationManager**
extends java.lang.Object

A Singleton which manages all the Organization functions. This class receives dispatched actions from the SOS Dispatcher and completes that action using objects internal to its subsystem. It also is in charge of interacting with the SOS Data Store Façade directly. Part of the role of this class is to parse front-end format data (e.g., JSON-String description of new Organization) and calling the appropriate functions on other classes according to that data. Another job of this class is to manage Role creation and assignment, as well as mediate the modification of data in an Organization, and of Event hosting.

Constructors

OrganizationManager

protected **OrganizationManager**()

A protected or private constructor ensures that no other class has access to the Singleton.

Methods

grantRole

public void **grantRole**(int userId,
int orgId, int[] privIds)

Grants a number of privileges to a User for a given Organization.

Parameters:

userId - the unique id of the User
orgId - the unique id of the Organization
privIds - the unique ids of the Privileges given to the User.

---

instance

public static OrganizationManager **instance**()

Returns:

The unique instance of this class.

Package security

Class Summary

AccessManager

A Singleton dealing with access control actions.

PasswordManager

A Singleton which deals with password control actions.

---

security

Class AccessManager

java.lang.Object
|
+--security.AccessManager

< Constructors > < Methods >

---

public class **AccessManager**
extends java.lang.Object

A Singleton dealing with access control actions. It implements most of the back-end side of the access policy for SOS and host the relevant Enumerations for access permissions and other privileges. It also must be called to do checks on the relevant actions, such as creating events, deleting profiles, etc.

Constructors

AccessManager

protected **AccessManager**()

A protected or private constructor ensures that no other class has access to the Singleton.

Methods

CheckPrivileges

public boolean **CheckPrivileges**()

Returns:

The result of privilege check for the current user class.

---

instance

public static AccessManager **instance**()

Returns:

The unique instance of this class.

---

security

Class PasswordManager

java.lang.Object
|
+--security.PasswordManager

< Constructors > < Methods >

---

public class **PasswordManager**
extends java.lang.Object

A Singleton which deals with password control actions. It implements most of the back-end side of the password policy for SOS, including resolving passwords and checking the input password against the database.

Constructors

PasswordManager

protected **PasswordManager**()

The constructor could be made private to prevent others from instantiating this class. But this would also make it impossible to create instances of PasswordManager subclasses.

Methods

EncryptPassword

public static java.lang.String **EncryptPassword**(java.lang.String password)

Parameters:

password - is a String to be validated

Returns:

will return an encrypted version of the password as a String

ValidateLogInCredentials

public static boolean **ValidateLogInCredentials**(java.lang.String username,
java.lang.String pwd)

Parameters:

username - is the user name for log in pwd -
is the user's password for log in

Returns:

is the validation of the login credentials

ValidatePassword

public static boolean **ValidatePassword**(java.lang.String password)

Parameters:

password - as a String to be validated

Returns:

is true if password successfully validates

---

instance

public static PasswordManager **instance**()

Returns:

The unique instance of this class.

Package sosInterface

Class Summary

SOSDispatcher

A Command class which propagates the front-end requests to their specific target controllers.

SOSServer

SOSServer communicates with the front-end for creation of events.

SOSSessionManager

SOSSessionManager keeps track of each session for every user.

---

sosInterface

Class SOSDispatcher

java.lang.Object
|
+--sosInterface.SOSDispatcher

< Constructors > < Methods >

---

public class **SOSDispatcher**
extends java.lang.Object

A Command class which propagates the front-end requests to their specific target controllers. The requests messages which are parsed and pre-processed by the SOS Server then are used to call an appropriate dispatch from the SOS Dispatcher, which is in charge of directly calling all other controllers. Instead of being their own classes, each subcommand is defined in terms of parametrizations of the dispatch function within the SOS Dispatcher. Internally, SOS Dispatcher also keeps track of these requests and stores them in the Database

Constructors

SOSDispatcher

public **SOSDispatcher**()

Methods

Dispatch

public void **Dispatch**()

The method for dispatching events.

GetAllEvents

public java.util.ArrayList **GetAllEvents**()

This method returns all of the events.

Returns:

is an ArrayList contain of Strings of events.

---

sosInterface

Class SOSServer

java.lang.Object
|
+--sosInterface.SOSServer

< Constructors > < Methods >

---

public class **SOSServer**
extends java.lang.Object

SOSServer communicates with the front-end for creation of events. Also it is held responsible for managing user sessions and keeping track of them, as well as dispatching messages through the system.

Constructors

SOSServer

protected **SOSServer**()

The constructor could be made private to prevent others from instantiating this class. But this would also make it impossible to create instances of SOSServer subclasses.

Methods

CreateEvent

public static boolean **CreateEvent**(java.lang.String event)

This method creates an event and stores its data.

Parameters:

event - is a String coming from the front-end including a JSON which stores all of the event details, including name, type, location, etc.

Returns:

is true if event is successfully created and false otherwise.

ParseMessage

public static java.lang.String **ParseMessage**(java.lang.String jsonString)

This method parses a message coming from the front-end which supposed to be in JSON format.

Parameters:

jsonString - is a String coming from the front-end including the JSON

Returns:

is the parsed message

---

instance

public static SOSServer **instance**()

Returns:

The unique instance of this class.

---

send

public static void **send**(java.lang.String action,
java.lang.Object payload)

Dispatch an action with the parameters defined in the payload.

Parameters:

action - the action to be dispatched. payload
- the payload of the action.

---

sosInterface

Class SOSSessionManager

java.lang.Object

|
+--sosInterface.SOSSessionManager

< Constructors > < Methods >

---

public class **SOSSessionManager**
extends java.lang.Object

SOSSessionManager keeps track of each session for every user. It can gather information about the current session, update the session, or even destroy it.

Constructors

SOSSessionManager

protected **SOSSessionManager**()

The constructor could be made private to prevent others from instantiating this class. But this would also make it impossible to create instances of SOSSessionManager subclasses.

Methods

DestroySession

public static boolean **DestroySession**(java.lang.String sessionID)

This method destroys a given user session

Parameters:

sessionID - is the ID for the session to be destroyed

Returns:

is true if DestroySession is successful and false otherwise.

GetCurrentSession

public static java.lang.String **GetCurrentSession**()

This method returns the information regarding a current live session

Returns:

is a String containing current session's ID.

GetSessionInformation

public static java.lang.String **GetSessionInformation**(java.lang.String sessionID)

This method returns the information regarding a given session.

Parameters:

sessionID - is the ID for the desired session

Returns:

is a String containing current session's information.

LogOutUser

public static boolean **LogOutUser**()

This method logs out the user from their current session.

Returns:

is true if Logout is successful and false otherwise.

UpdateSessionInformation

public static boolean **UpdateSessionInformation**(java.lang.String data,
java.lang.String sessionID)

This method updates the given session with the modified data

Parameters:

sessionID - is the ID of the session to be updated
data - is the modified data to be updated on the given session

Returns:

is true if update information is successfull and false otherwise.

---

instance

public static SOSSessionManager **instance**()

Returns:

The unique instance of this class.

Package user

Class Summary

NewUserBuilder

A Builder which creates new User objects.

User

A run-time representation of a User persistent object.

UserLoader

A class which creates a User object from a database-format User object (e.g., a SQL Table entry for User).

UserManager

A Singleton class which managers all the User functions.

UserUpdater

A class which deals with User modifications.

---

user

Class NewUserBuilder

java.lang.Object
|
+--user.NewUserBuilder

< Constructors > < Methods >

---

public class **NewUserBuilder**
extends java.lang.Object

A Builder which creates new User objects. It is used to decouple the parts of the process of creating a new User from the actual User class, which is intended to only be a data wrapper class which can be easily parsed into the database format. Namely, this class implements the checks and validations necessary to create a valid User and will reject invalid ones. As part of this validation, it must interact with the SOS Security System classes that implement the password and access policies.

Constructors

NewUserBuilder

public **NewUserBuilder**()

Creates a new NewUserBuilder to instantiate the new User.

Methods

BuildUser

public User **BuildUser**()

Finalizes the User creation and returns it.

Returns:

returns the final User.

ValidateCredentials

protected boolean **ValidateCredentials**()

Checks the current User, returning True if it is valid so far and False otherwise.

Returns:

True, if the Event is valid so far. False if otherwise.

---

setEmail

public NewUserBuilder **setEmail**(java.lang.String email)

Adds a email value to the current User.

Parameters:

email - the value to be added.

Returns:

the current builder.

---

setName

public NewUserBuilder **setName**(java.lang.String name)

Adds a name value to the current User.

Parameters:

name - the value to be added.

Returns:

the current builder.

---

setPassword

public NewUserBuilder **setPassword**(java.lang.String password)

Adds a password value to the current User.

Parameters:

password - the value to be added.

Returns:

the current builder.

---

setPrivacy

public NewUserBuilder **setPrivacy**(java.lang.String privacy)

Adds a privacy value to the current User.

Parameters:

privacy - the value to be added.

Returns:

the current builder.

---

setUsername

public NewUserBuilder **setUsername**(java.lang.String username)

Adds a username value to the current User.

Parameters:

username - the value to be added.

Returns:

the current builder.

---

user

Class User

java.lang.Object
|
+--user.User

< Constructors >

public class **User**
extends java.lang.Object

A run-time representation of a User persistent object. This class is used as an intermediary for creation, retrieval, and modification of User data within the Java code (and the JVM). It is encodable (or serializable) to a database format (e.g., SQL Entry).

Constructors

User

protected **User**()

Constructs a new User class. Called through the UserBuilder class. Attribute assignations are done through protected scope.

user

Class UserLoader

java.lang.Object
|
+--user.UserLoader

< Constructors > < Methods >

public class **UserLoader**
extends java.lang.Object

A class which creates a User object from a database-format User object (e.g., a SQL Table entry for User). This class decouples the parsing from the database to the system logic from the UserManager class and can be extended to include internal checks for data integrity purposes.

Constructors

UserLoader

public **UserLoader**()

Methods

LoadOrganization

public static User **LoadOrganization**(java.lang.String sqlEntry)

Creates a User from a database-format entry.

Parameters:

sqlEntry - a sql entry for the given organization.

Returns:

a User object with the given attributes.

---

user

Class UserManager

java.lang.Object
|
+--user.UserManager

< Constructors > < Methods >

---

public class **UserManager**
extends java.lang.Object

A Singleton class which managers all the User functions. This class receives dispatched actions from the SOS Dispatcher and completes that action using objects internal to its subsystem. It also is in charge of interacting with the SOS Data Store Façade directly. Part of the role of this class is to parse front-end format user data (e.g., JSON-String defining a new User) and calling the appropriate functions on the other classes according to that data. It also is in charge of encoding a User object into database format objects (e.g., SQL Table entry for User).

Constructors

UserManager

protected **UserManager**()

A protected or private constructor ensures that no other class has access to the Singleton.

Methods

ChangeUserDetails

public void **ChangeUserDetails**(User user,
java.util.Map change)

Updates a User object with the given changes. Done through the UserUpdater class.

Parameters:

user - the User that will be updated.
change - a Map where the key is the variable name and the value is the update.

CreateNewProfile

public User **CreateNewProfile**(java.lang.String jsonString)

Creates a new User from a json User description. Done by calling the NewUserBuilder class.

Parameters:

jsonString - the JSON object describing the new User.

Returns:

a User object with the given attributes.

LoadUser

public User **LoadUser**(java.lang.String sqlEntry)

Creates a User from a database-format entry. Done by calling the UserLoader class.

Parameters:

sqlEntry - a sql entry for the given user.

Returns:

a User object with the given attributes.

---

instance

public static UserManager **instance**()

Gives the instance of the UserManager, or creates one if none exists.

Returns:

the unique instance of this class.

---

user

Class UserUpdater

java.lang.Object
|
+--user.UserUpdater

< Constructors > < Methods >

public class **UserUpdater**
extends java.lang.Object

A class which deals with User modifications. User modifications are done on the system logic-level User object first and are only finalized once they are stored to the database. The UserUpdater decouples these modifications from the UserManager class and from the User class itself and implements checks and validations in the same way that NewUserBuilder does. It also ensures that every modification to the User class is saved to the SOS Data Store.

Constructors

UserUpdater

public **UserUpdater**()

Methods

ChangeUser

public void **ChangeUser**(User user,
java.util.Map update)

Updates a User object with the given changes.

Parameters:

user - the User that will be updated.
change - a Map where the key is the variable name and the value is the update.

7.5 Appendix E – Diary of Meetings

7.5.1 October 7, 2019

When and Where	Role
Date: 10/7/19	**Primary Facilitator:** Teriq
Start: 2:30 pm	**Timekeeper:** Yovanni
End: 3:30 pm	**Minute Taker:** Anthony
Room: GL 693	**Attending:** Armando, Kian (late), Teriq, Anthony, Yovanni (late)

1. Status

   First meeting for deliverable 2. The focus of the meeting was generating tasks from the second deliverable document and assigning the tasks that are immediately available.

2. Discussion

   Task Decomposition:
   - Read the second deliverable. **All**
   - Cover Page – Refine from SRD
   - Abstract – Something to do at the end.
   - Table Of Contents – Something to do at the end.
   - Introduction – Refine from SRD
   - Section 1.1 – Refine from SRD
   - Section 1.2 Requirements
   - Functional Requirements - **Kian**
   - Non-Functional Requirements - **Kian**
   - Section 1.3 Minimal edits to the SRD.
   - Section 1.4 Something to do at the end.
   - Section 1.5 Something to do at the end.
   - Section 2
   - 2.1 Overview – Identify the different subsystems. Depends on 2.2
   - 2.2 *Subsystem decomposition is the 1st task.* **All**
   - 2.3 Hardware and Software Mapping
   - 2.4 Persistent Data Management. **Teriq & Anthony**
   - 2.5 Security Management. **Armando**
   - Section 3
   - Introduction
   - 3.1 Class diagrams for subsystems that will be implemented (no details). Depends on 2.2.
   - 3.2 Depends on 2.2
   - 3.4.1 Cannot do yet.
   - 3.4.2 Cannot do yet. Object Constraint Language (OCL)
   - 4 Glossary – Something to do at the end
   - Appendices A – Refine from SRD
   - Appendices B – Refine from SRD
   - Appendices C – Create new class diagram for all the subsystems that will be implemented.

- Appendices D – Javadoc on coding
- Appendices E – In progress.

3. Wrap Up

- Teriq, Anthony and Yovanni will continue to conduct research on the back-end development.
- Started doing subsystem decomposition.

7.5.2 October 14, 2019

When and Where	Role
Date: 10/14/2019	**Primary Facilitator:** Armando
Start: 2:00 pm	**Timekeeper:** Yovanni
End: **3**:00 pm	**Minute Taker:** Anthony
Room: GL 595A	**Attending:** Armando, Kian, Anthony, Yovani

1. Status

   Update regarding status of project contributions. The focus of the meeting was to discuss the different architectural patterns and identify which of the two the team would use in our product. Tasks needed to be assigned.

2. Discussion

   The discussions started with Armando which said that he had no updates and that he would work on the security system as soon as he got the chance to. Anthony then talked about Teriq and his development of the ER diagram. He queried about the structure of the data base and spurred lively debate on certain attributes for persistent data. Anthony also talked about the retrieval of address locations from the Google Location API. Kian status was that he had worked on the front-end and that he had investigated an API called Springboot that applied encryption on the front-end of the SOS website. He also talked about the google maps module, the container component and the API key. Lastly, Yovanni claimed that he had started to work on the tasks that had been assigned to him in the previous meeting and that he was almost finished with them.

   After the discussion and status of the work of all the team-members we discussed the different architectures within the team, and we decided that the two architectures ideal for our system would be 3-tier architecture and repository architecture. The 3-tier architecture would serve as our primary architecture and the repository architecture would serve as the secondary architecture use to store and access information from our database. After deciding our architectural patterns, the meeting was disbanded.

3. Wrap Up
   - The architectural patterns for our system were decided.
   - Armando will work on the security section.
   - Anthony will keep working on implementing the database and refining ER diagram.
   - Kian will finish the functional requirements in section 1.
   - Yovanni will work on section 1, specifically the nonfunctional requirements.

7.5.3 October 21, 2019

When and Where	Role
Date: 10/21/19	**Primary Facilitator:** Teriq
Start: 2:30 pm	**Timekeeper:** Yovanni
End: 3:30 pm	**Minute Taker:** Anthony
Room: ECS 243	**Attending:** Armando, Kian, Teriq, Anthony, Yovanni

1. Status

   Update regarding tasks assigned previous week. Some members have successfully completed their tasks while others must revisit their works due to slight misconceptions found. Start decomposing the system into subsystems.

2. Discussion

   Armando, Yovanni and Kian all finished their tasks successfully. Section 1 of the Design Document is 35% complete and Armando completed the security section for the Design Document. Teriq and Anthony made a slight mistake while creating the ER Diagram as they did not use a validation software so it must be redone. Teriq took responsibility of doing the data dictionaries for the persistent data section.

   Afterwards, the team discussed how the system should be effectively decomposed. The team decided that the system should be decomposed into three layers which includes the Presentation layer, Logic layer and the Storage layer. Within these major subsystems the team identified key partitions that would ensure high cohesion and low coupling between different subsystems.

   The team decided to make 7 different subsystems in the logic layer including SOS server, SOS session manager, SOS Dispatcher, User Management, Event Management, Organization Management and Security Management. The first three subsystems are specialized to retrieve information from the user, keep track of the current status of the system and send data back to the system. The rest of the subsystems found within the Logic layer are to delegate certain operations for the persistent objects that exist within our system which include events, organizations and users.

3. Wrap Up
   - Teriq will complete the data dictionaries for the persistent data section of the design document.
   - Anthony will revisit the ER Diagram and re-do it on STAR UML.
   - Armando and Yovanni will work on the hardware and software mapping of the SOS system.
   - Kian will continue to work on the front end of the SOS system and on Section 1 of the Design Document.

7.5.4 October 28, 2019

When and Where	Role
Date: 10/28/19	**Primary Facilitator:** Teriq
Start: 2:30 pm	**Timekeeper:** Yovanni
End: 3:30 pm	**Minute Taker:** Anthony
Room: ECS 243	**Attending:** Armando, Kian, Teriq, Anthony, Yovanni

1. Status

Update regarding tasks assigned previous week. Member have all successfully completed their tasks. This week we start formatting the Design Document and assigning additional tasks to team members.

2. Discussion

Armando and Yovanni completed the hardware and software mapping of the SOS system. They presented it to the team and after a short debate we decided that it was an adequate representation of how SOS system would be launched. Afterwards, Kian gave an update of what he had completed in the front end of the system and informed us that he had completed Section 1 of the Design Document.

Teriq completed approximately half of the data dictionaries because he was waiting for Anthony to finalize the ER diagram to ensure that both data dictionaries and ER diagram reflected the persistent objects found in our system. Anthony presented the finalize version do on STAR-UML to the team and there was a consensus that made it the final representation of the system database design.

Afterwards, the team decided to go through an overview of the design patterns that our system may have, and we decided to start creating the class diagrams that would represent those patterns. In addition the team also started thinking about the way the minimal class diagrams would be connected and looking at the requirements to complete section 3 of the design document.

3. Wrap Up

- Teriq needs to finish the data dictionaries as soon as possible.
- Anthony needs to implement the ER diagram into MySQL.
- Armando needs to complete the minimal class diagrams for all the subsystems.
- Kian needs to continue working on the front end as well as the structuring of the design document.
- Yovanni need to start brainstorming and researching the OCL statements for each major subsystem.

7.5.5 November 4, 2019

When and Where	Role
Date: 11/04/19	**Primary Facilitator:** Teriq
Start: 2:30 pm	**Timekeeper:** Yovanni
End: 3:30 pm	**Minute Taker:** Anthony
Room: ECS 243	**Attending:** Armando, Kian, Teriq, Anthony, Yovanni

1. Status

   Update on the status of the tasks that were assigned in the previous week. Assignment of remaining tasks to the team members. Update on the status of the system and the completion of the design document.

2. Discussion

   In this meeting the team decided to first see how much was missing in the design document and what was currently implemented. All of section 1 was complete. All of section 2 had been complete. Armando had finished the overview of the class diagrams for the subsystems along with their descriptions. Yovanni had generated the OCL for the major control objects in each major subsystem. Teriq had finished the data dictionaries and started writing the java class interfaces for the main control object in each subsystem.

   Anthony had finished implementing all of the tables that were specified in the persistent data section in MySQL and he started working on the state machine and object interaction sections of section 3 and was about 25% done. Kian presented his additions in the front end and decided to help Armando with section 3.4.1 and Appendix C.

   The missing tasks for the design document are Appendix E which is currently being finished. Appendix A which is found on the SRD. The approval page, references, glossary and introduction to the object design chapter. Armando decided to take responsibility for these remaining roles. We decided that everything should be finished by November 08, 2019 to allow time for revision.

3. Wrap Up
   - Teriq needs to finish the java class interface for the main control object in each subsystem.
   - Anthony needs to finish the state machine and object interactions.
   - Armando, Yovanni and Kian must work together to finish the Detailed Class Design section along with Appendix C.

7.5.6 November 8, 2019

When and Where	Role
Date: 11/08/19	**Primary Facilitator:** Teriq
Start: 11:30 pm	**Timekeeper:** Yovanni
End: 04:30 pm	**Minute Taker:** Anthony
Room: ECS 243	**Attending:** Armando, Kian, Teriq, Anthony, Yovanni

1. Status

   Update with the tasks assigned last week. Proofread the document and revisit all the charts found in the design document to ensure correctness of the deliverable.

2. Discussion

   All members, except Anthony, seem to have finished their sections of the report. Anthony finished object design but seemed to be struggling with the state machine diagram. After reviewing Teriq's interface implementation we decided to readjust some of the code that he had written. Armando, Kian and Yovanni had finished the section that were assigned to them.

   After giving the update of our progress the team spent some time reading the document and debating on the correctness of the ideas expressed within the document. More importantly we made sure that the document was in the correct format and that the reader could find things easily and view diagrams with ease.

   By the time the meeting was coming to a halt Teriq had finished his java class interfaces and had published them in the document. Anthony after help from everyone on the team created a state machine diagram for the overall system and the main control object in each major subsystem. The Design Document was about 90% complete and we decided as a team that over the long weekend we would email the professor about our questions regarding the correctness of our approach and proofread the content found within it.

3. Wrap Up
   - Proofread the document.
   - Ask the professor questions about the confusions found in the document.
   - Turn in the document.