saga

Project Management Application

Low-Level Design (LLD) Document

Version 1.2

July 10, 2020

Prepared By Group 2

Tulin Akbulut
Alex Alibrandi
Michael Jaouhari
Matt Merle
Brandon Miracle
Melissa Ross

Revision Tables

Revisions

Version	Primary Author(s)	Description of Version	Date Completed
0.0	M. Ross	Outline	07/02/2020
1.0	B. Miracle	Outline Detail	07/06/2020
1.1	A. Alibrandi T. Akbulut M. Jaouhari	Section Content (Draft)	07/09/2020
1.2	M. Ross A. Alibrandi B. Miracle	Final Draft	07/10/2020

Review and Approval

Approving Party	Version Approved	Signature	Date
B. Miracle	0.0	RBM	07/03/2020
B. Miracle	1.0	RBM	07/07/2020
M. Ross	1.1	MMR	07/09/2020
M. Ross	1.2	MMR	07/10/2020

Low-Level Design Document Review History

Reviewer	Version Reviewed	Signature	Date
M. Ross	1.1	MMR	07/10/2020
M. Ross	1.2	MMR	07/10/2020

Table of Contents

1. Introduction

1.1 Purpose

Saga is a project management application that will allow its users to create a product without the hassle of using numerous other applications to achieve their goal. Saga aims to ease the process of project management by combining the necessary tools we believe is needed in successful project development. These tools include Kanban boards to visualize work and optimize the flow of work throughout the team. Another tool we will include is an analytical reporting tool that will generate reports such as status reports, risk reports, variance reports, and resource reports.

1.2 Scope of this Document

This low-level design document will refine the application's design by going into detail on the following:

- Object-oriented design and class hierarchy
- Database design
- Verification and validation testing

This document will provide details on the classes and tables in this application, including diagrams and pseudocode to guide programmers. This LLD is intended to be read by developers and designers who are developing this application. After reading the contents of this document, developers will be able to program Saga.

1.3 Definitions, Acronyms and Abbreviations

- GUID: Globally Unique Identifier
- HLD: High Level Design
- Int: Shortened version of Integer
- KPI: Key Performance Indicator
- LLD: Low-Level Design
- PBKDF2: Password-Based Key Derivation Function 2
- SDLC: Software Development Life Cycle
- UI: User Interface
- UML: Unified Modeling Language, language for software modeling

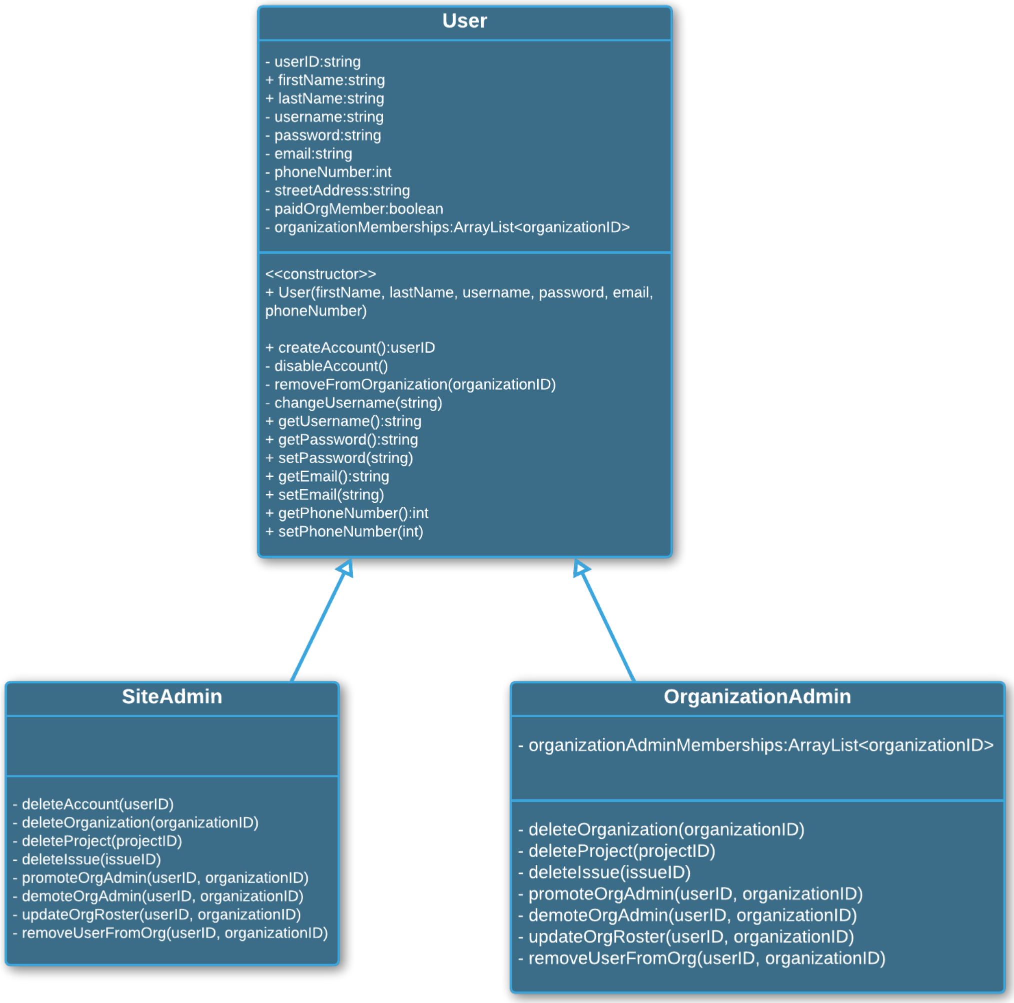
2. Object-Oriented Design

2.1 Overview

The object-oriented design for Saga will define the specific classes that the application will use, including their designated attributes, methods and events. It will also provide insight into the inheritance relationships and hierarchies for these classes, their refinements, the generalizations created and their object compositions.

2.2 Classes

2.2.1 Users


User
- userID:string
+ firstName:string
+ lastName:string
- username:string
- password:string
- email:string
- phoneNumber:int
- streetAddress:string
- paidOrgMember:boolean
- organizationMemberships:ArrayList<organizationID>
<<constructor>>
+ User(firstName, lastName, username, password, email, phoneNumber)
+ createAccount():userID
- disableAccount()
- removeFromOrganization(organizationID)
- changeUsername(string)
+ getUsername():string
+ getPassword():string
+ setPassword(string)
+ getEmail():string
+ setEmail(string)
+ getPhoneNumber():int
+ setPhoneNumber(int)
SiteAdmin
- deleteAccount(userID)
- deleteOrganization(organizationID)
- deleteProject(projectID)
- deleteIssue(issueID)
- promoteOrgAdmin(userID, organizationID)
- demoteOrgAdmin(userID, organizationID)
- updateOrgRoster(userID, organizationID)
- removeUserFromOrg(userID, organizationID)
OrganizationAdmin
- organizationAdminMemberships:ArrayList<organizationID>
- deleteOrganization(organizationID)
- deleteProject(projectID)
- deleteIssue(issueID)
- promoteOrgAdmin(userID, organizationID)
- demoteOrgAdmin(userID, organizationID)
- updateOrgRoster(userID, organizationID)
- removeUserFromOrg(userID, organizationID)

Class name: Users	
Description: The Users class will be responsible for defining and identifying a human user within the system.	
Attributes	
String userID;	**Attribute Description**
	This is a declaration of the user's identification.
	Program Description Language
	Private String userID
String firstName;	**Attribute Description**
	This is a declaration of the user's first name.
	Program Description Language
	Public String firstName;
String lastName;	**Attribute Description**
	This is a declaration of the user’s last name.
	Program Description Language
	Public String lastName;
String username;	**Attribute Description**
	This is a declaration of the user’s username.
	Program Description Language
	Private String username;

String password;	**Attribute Description**
	This is a declaration of the user’s password.
	Program Description Language
	Private String password;
String email;	**Attribute Description**
	This is a declaration of the user’s email address.
	Program Description Language
	Private String email;
Int phoneNumber;	**Attribute Description**
	This is a declaration of the user’s phone number.
	Program Description Language
	Private Int phoneNumber;
String streetAddress;	**Attribute Description**
	This is a declaration of the user’s street address.
	Program Description Language
	Private String streetAddress;
Boolean paidOrgMember;	**Attribute Description**
	This is a boolean value defining if the user is a member of a paid organization.
	Program Description Language
	Private Boolean paidOrgMember;

ArrayList organizationMemberships;	**Attribute Description**
	This is an array list of organization memberships of which the user is a member.
	Program Description Language
	Private ArrayList organizationMemberships;
Methods	
User()	**Method Description**
	A constructor method that collects firstName, lastName, username, password, email, and phoneNumber attributes.
	Program Description Language

	Public User(){
	 Create new User object
	 Initialize object with input data
	 Assign GUID to object as *userID*
	}

createAccount()	**Method Description**
	This method will be used to create a users account.
	Program Description Language

	Public createAccount(){
	  Gather user input data
	  Call User constructor, passing in data
	}

disableAccount()	**Method Description**
	This is a method to disable a user's account.
	Program Description Language

	Private disableAccount(){
	  Locate account based on which User calls method
	  Disable account
	}

removeFromOrganization(organizationID)	**Method Description**
	This is a method to remove a user from an organization.
	Program Description Language

	Private removeFromOrganization(organizationID){
	  Locate organization based upon *organizationID*
	  Remove calling User from Organization
	}

changeUsername(string)	**Method Description**
	This is a method to change a user's username.
	Program Description Language

	Private changeUsername(string){
	  Gather user input data
	  Change *username* based on string input
	}

String getUsername()	**Method Description**
	This method gets the correct username.
	Program Description Language

	Public getUsername(){
	  Get *username* attribute of object
	  Return *username* attribute}

String getPassword()	**Method Description**
	This method gets the correct user password.
	Program Description Language

	Public getPassword(){
	  Get *password* attribute of object
	  Return *password* attribute
	}

String setPassword(string)	**Method Description**
	This method sets the user password based on the calling User object.
	Program Description Language

	Public setPassword(string){
	  Access *password* attribute of object
	  Set *password* = string argument
	}

String getEmail()	**Method Description**
	This method gets the User object's email attribute.
	Program Description Language

	Public getEmail(){
	  Get *email* attribute of object
	  Return *email* attribute
	}

String setEmail(string)	**Method Description**
	This method sets the User object's email attribute.
	Program Description Language

	Public setEmail(string){
	   Access *email* attribute of object
	   Set *email* = string argument}

Int getPhoneNumber()	**Method Description**
	This method gets the User object's phone number attribute.
	Program Description Language

	Public getPhoneNumber(){
	  Get *phoneNumber* attribute of object
	  Return *phoneNumber* attribute
	}

Int setPhoneNumber(int)	**Method Description**
	This method sets the User object's phone number attribute.
	Program Description Language

	Public setPhoneNumber(int){
	  Access *phoneNumber* attribute of object
	  Set *phoneNumber* = string argument
	}

Subclass name: SiteAdmin	
Description: The SiteAdmin class will consist of methods defining all the capabilities of a site admin.	
Methods	
deleteAccount(userID)	**Method Description**
	This method will make it so a designated site admin can delete any account using a user's unique identifier.
	Program Description Language

	Private deleteAccount(userID){
	  Locate account based upon *userID*
	  Delete users account
	}

deleteOrganization(organizationID)	**Method Description**
	This method will make it so a designated site admin can delete any organization based upon an organization's unique identifier.
	Program Description Language

	Private deleteOrganization(organizationID){
	  Locate organization based upon *organizationID*
	  Delete organization
	}

deleteProject(projectID)	**Method Description**
	This method will make it so a designated site admin can delete any project in an organization based upon a projects unique identifier
	Program Description Language

	Private deleteProject(projectID){
	  Locate project based upon *projectID*
	  Delete project
	}

deleteIssue(issueID)	**Method Description**
	This method will make it so a designated site admin can delete any issue within a project based upon an issue's unique identifier.
	Program Description Language

	Private deleteIssue(issueID){
	  Locate issue based upon *issueID*
	  Delete issue
	}

promoteOrgAdmin(userID, organizationID)	**Method Description**
	This method will make it so a designated site admin can promote a user to org admin status based upon the unique user and organization identifiers.
	Program Description Language

	Private promoteOrgAdmin(userID,organizationID){
	  Locate organization by *organizationID*
	  Locate user in organization by *userID*
	  Add userID to Organization’s OrgAdmin roster
	}

demoteOrgAdmin(userID, organizationID)	**Method Description**
	This method will make it so a designated site admin can demote any OrgAdmin to normal user status based upon the unique user and organization identifiers.
	Program Description Language

	Private demoteOrgAdmin(userID,organizationID){
	  Locate organization by *organizationID*
	  Locate user by *userID*
	  Remove *userID* from Organization's OrgAdmin roster
	  Ensure *userID* is still listed in Organization's member roster
	}

updateOrgRoster(userID, organizationID)	**Method Description**
	This method will make it so any site admin can make changes to any organization's roster based upon the unique user and organization identifier.
	Program Description Language

	Private updateOrgRoster(userID, organizationID){
	  Locate organization by *organizationID*
	  Access org roster
	  Locate user by *userID*
	  Add user to org roster
	}

removeUserFromOrg(userID, organizationID)	**Method Description**
	This method will make it so any designated site admin can remove a user for an organization based upon the unique user and organization identifier.
	Program Description Language

	Private removeUserFromOrg(userID,organizationID){
	  Locate organization by *organizationID*
	  Access org roster
	  Locate user based on *userID*
	  Remove user from org roster
	}

Subclass name: OrganizationAdmin	
Description: This class will help designate organization admin memberships and the capabilities of an organization admin.	
Attributes	
ArrayList organizationAdminMemberships	**Attribute Description**
	This attribute is an array list consisting of all organizations to which a user belongs as an OrgAdmin.
	Program Description Language
	Private ArrayList organizationAdminMemberships;
Methods	
deleteOrganization(organizationID)	**Method Description**
	This method will allow the designated organization admin to delete an organization of which they are an OrgAdmin based upon the unique organizationID.
	Program Description Language

	Private deleteOrganization(organizationID){
	  Locate organization based upon *organizationID*
	  Delete organization
	}

deleteProject(projectID)	**Method Description**
	This method will allow the designated organization admin to delete any project created within their organization based upon the unique project identifier.
	Program Description Language

	Private deleteProject(projectID){
	  Locate project based on *projectID*
	  Delete project
	}

deleteIssues(issueID)	**Method Description**
	This method will allow the designated organization admin to delete any issues in a project based upon the designated unique issue identifier.
	Program Description Language

	Private deleteIssues(issueID){
	  Locate issue based on *issueID*
	  Delete issue
	}

promoteOrgAdmin(userID, organizationID)	**Method Description**
	This method will allow the designated organization admin to promote any user in their organization to OrgAdmin status based upon the unique user, and organization identifiers.
	Program Description Language

	Private promoteOrgAdmin(userID, OrganizationID){
	  Locate organization by *organizationID*
	  Locate user based on *userID*
	  Promote user to OrgAdmin
	}

demoteOrgAdmin(userID, organizationID)	**Method Description**
	This method will allow an organization admin to demote any user appointed org admin status back to standard user status based upon the unique user, and organization identifiers
	Program Description Language

	Private demoteOrgAdmin(userID, organizationID{
	    Locate organization by organizationID
	    Locate user based on userID
	    Demote OrgAdmin
	}

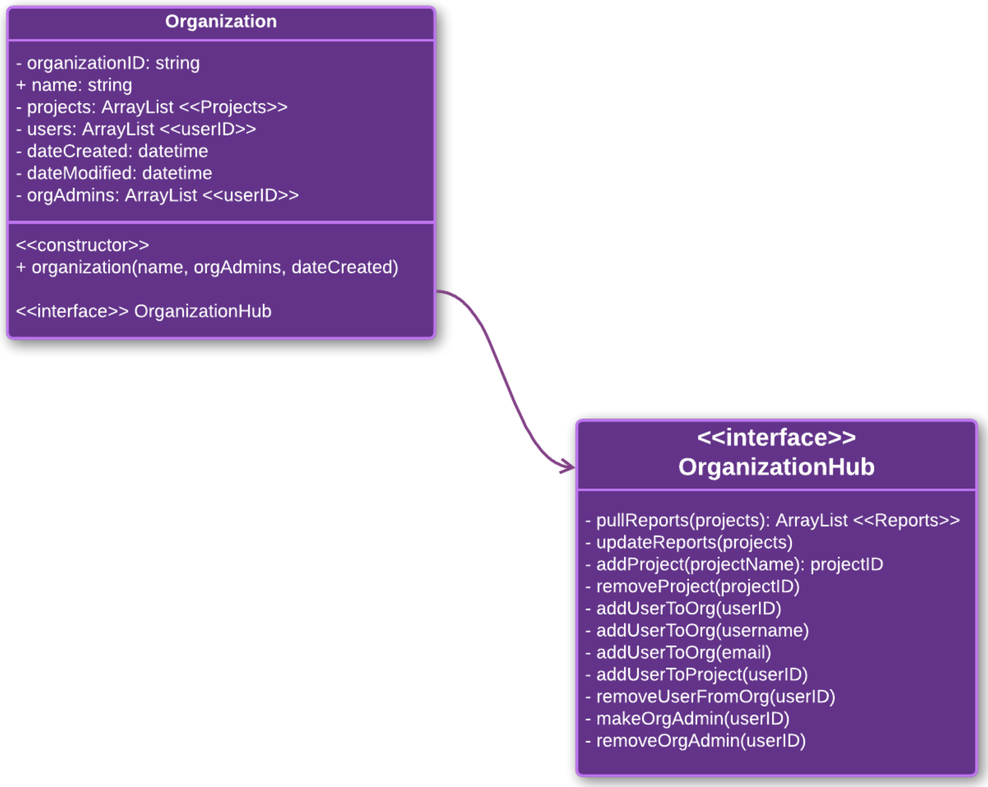
updateOrgRoster(userID, organizationID)	**Method Description**
	This method will allow the designated organization admin to update any organization roster they have org admin status in, based upon the unique user and organization identifiers.
	Program Description Language

	Private updateOrgRoster(userID, organizationID){
	    Locate organization based on organizationID
	    Locate user based on userID
	    Add user to OrgRoster
	}

removeUserFromOrg(userID,organizationID)	**Method Description**
	This method will allow the designated organization admin to remove any user from an organization they have org admin status in, based upon the unique user and organization identifiers.
	Program Description Language

	Private removeUserFromOrg(userID,organizationID){
	    Locate organization based on organizationID
	    Locate user based on userID
	    Remove user from org
	}

2.2.2 Organization


Organization
- organizationID: string
+ name: string
- projects: ArrayList <<Projects>>
- users: ArrayList <<userID>>
- dateCreated: datetime
- dateModified: datetime
- orgAdmins: ArrayList <<userID>>
<<constructor>>
+ organization(name, orgAdmins, dateCreated)
<<interface>> OrganizationHub
<<interface>>
OrganizationHub
- pullReports(projects): ArrayList <<Reports>>
- updateReports(projects)
- addProject(projectName): projectID
- removeProject(projectID)
- addUserToOrg(userID)
- addUserToOrg(username)
- addUserToOrg(email)
- addUserToProject(userID)
- removeUserFromOrg(userID)
- makeOrgAdmin(userID)
- removeOrgAdmin(userID)

Class name: Organization	
Description: The Organization class will be responsible for defining organization information and managing actions that pertain to organizations.	
Attributes	
String organizationID;	**Attribute Description**
	This is the declaration of the organization's identification.
	Program Description Language
	Private String organizationID;

String name;	**Attribute Description**
	This is the declaration of the organization’s name.
	Program Description Language
	Public String name;
ArrayList projects;	**Attribute Description**
	This is an array list of Project objects that are owned by the organization.
	Program Description Language
	Private ArrayList projects;
ArrayList users;	**Attribute Description**
	This is an array list of userIDs from the Users in the Organization.
	Program Description Language
	Private ArrayList users;
Datetime dateCreated;	**Attribute Description**
	This is a declaration of the date and time the organization was created.
	Program Description Language
	Private Datetime dateCreated;
Datetime dateModified;	**Attribute Description**
	This is a declaration of the date and time the organization was last modified.
	Program Description Language
	Private Datetime dateModified;

ArrayList orgAdmins;	**Attribute Description**	
	This is an array list of userIDs from the organization admins for the Organization.	
	Program Description Language	
	Private ArrayList orgAdmins;	
Methods		
organization(name, orgAdmins, dateCreated);	**Method Description**	
	The constructor method for the Organization class that collects name, orgAdmins, and dateCreated attributes.	
	Program Description Language	

	Public organization(name, orgAdmins, dateCreated){
	  Create new Organization object
	  Initialize object with input values
	  Assign a GUID to *organizationID*
	}	

Interface name: OrganizationHub	
Description: The organization hub interface will contain the methods that document the capacity of organization customization under organization admin status.	
Methods	
Report pullReports(projects)	**Method Description**
	This method will gather reports for a specific project.
	Program Description Language

	Private Reports pullReports(projects){
	    Locate Report objects by *projects*
	    Add each Report object located to a new array list
	    Return the array list of Report objects
	}

updateReports(projects)	**Method Description**
	This method will update reports to their current status based upon the designated project in the projects array list.
	Program Description Language

	Private updateReports(projects){
	   Call *pullReports(projects)* method using *projects* to return Report objects
	   For each Report, call it's *refreshReport()* method
	}

projectID addProject(projectName)	**Method Description**
	This method will add a project to an organization and return the unique projectID for the project.
	Program Description Language

	Private projectID addProject(projectName){
	    Create project object, calling constructor
	    Call project by projectName
	    Add project to the organization's project list
	}

removeProject(projectID)	**Method Description**
	This method will remove a project from an organization based upon the unique project identifier.
	Program Description Language

	Private removeProject(projectID){
	    Locate project based on projectID
	    Remove project
	}

addUserToOrg(userID)	**Method Description**
	This method will add a new user to an organization based upon their unique user identifier.
	Program Description Language

	Private addUserToOrg(userID){
	    Locate user based on userID
	    Add user to org
	}

addUserToOrg(username)	**Method Description**
	This method will add a new user to an organization based upon their unique username.
	Program Description Language

	Private addUserToOrg(username){
	    Locate user based on username
	    Add user to org
	}

addUserToOrg(email)	**Method Description**
	This method will add a new user to an organization based upon their unique email address.
	Program Description Language

	Private addUserToOrg(email){
	    Locate user based on email address
	    Add user to org
	}

addUserToProject(userID)	**Method Description**
	This method will add a user to a project based upon their unique user identifier.
	Program Description Language
	Private addUserToProject(userID){ Locate user based on userID Add user to org}

removeUserFromOrg(userID)	**Method Description**
	This method will remove a user from an organization based upon their unique user identifier.
	Program Description Language

	Private removeUserFromOrg(userID){
	    Locate user based on userID
	    Remove user from org
	}

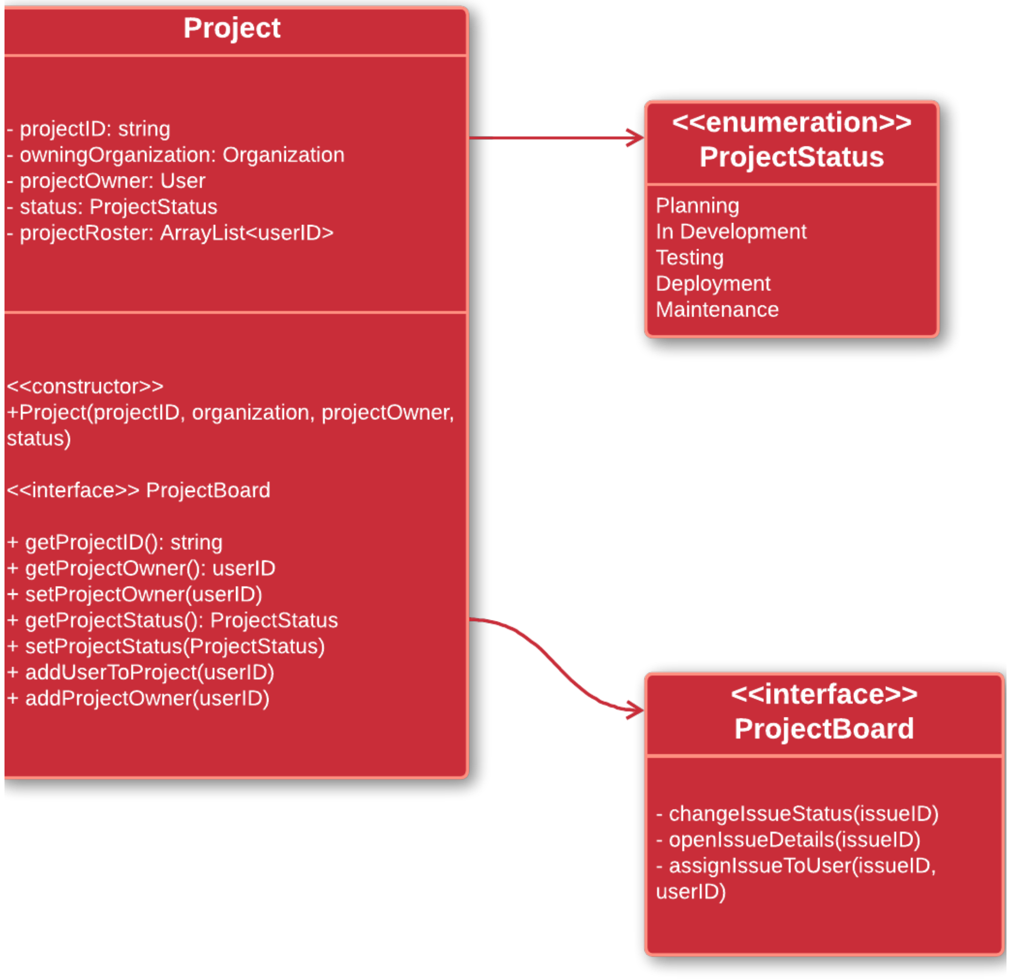
makeOrgAdmin(userID)	**Method Description**
	This method will allow an organization admin to promote a user of their organization to admin status in their organization based upon the unique user identifier.
	Program Description Language

	Private makeOrgAdmin(userID){
	    Locate user based on userID
	    Make user orgAdmin
	}

removeOrgAdmin(userID)	**Method Description**
	This method will allow an organization admin to remove another organization admin based upon the unique user identifier.
	Program Description Language

	Private removeOrgAdmin(userID){
	    Locate user based on userID
	    Remove orgAdmin status from user
	    Ensure user is still in org
	}

2.2.4 Project


Project
- projectID: string
- owningOrganization: Organization
- projectOwner: User
- status: ProjectStatus
- projectRoster: ArrayList<userID>
<<constructor>>
+Project(projectID, organization, projectOwner, status)
<<interface>> ProjectBoard
+ getProjectID(): string
+ getProjectOwner(): userID
+ setProjectOwner(userID)
+ getProjectStatus(): ProjectStatus
+ setProjectStatus(ProjectStatus)
+ addUserToProject(userID)
+ addProjectOwner(userID)
<<enumeration>> ProjectStatus
Planning
In Development
Testing
Deployment
Maintenance
<<interface>> ProjectBoard
- changeIssueStatus(issueID)
- openIssueDetails(issueID)
- assignIssueToUser(issueID, userID)

Class name: Project	
Description: The Project class will define and identify projects within an organization.	
Attributes	
String projectID;	**Attribute Description**
	This is a declaration of the project's identification.
	Program Description Language
	Private String projectID;

owningOrganization	**Attribute Description**
	This attribute is an object of class organization and describes who the organization belongs to.
	Program Description Language
	Private Organization owningOrganization;
projectOwner	**Attribute Description**
	This attribute is an object of class user and describes the owner of a project.
	Program Description Language
	Private User projectOwner;
status	**Attribute Description**
	This attribute is an object of the enumeration projectStatus and describes the status of a project.
	Program Description Language
	Private ProjectStatus status;
Methods	
Project(projectId, organization, projectOwner, status);	**Method Description**
	This constructor method builds a project using the projectID, organization, projectOwner, and status attributes.
	Program Description Language

	Public Project (projectId, organization, projectOwner, status){
	  Create new Project object
	  Initialize object with inputs
	  Assign GUID to *projectID*
	}

String getProject();	**Method Description**
	This method gets the projectID.
	Program Description Language

	Public getProject(){
	  Get *projectID* attribute of an object
	  Return *projectID* attribute
	}

getProjectOwner();	**Method Description**
	This method determines the project owner of a specific project.
	Program Description Language

	Public getProjectOwner(projectID){
	  Get *projectOwner* attribute of an object
	  Return *projectOwner* attribute
	}

setProjectOwner(userID);	**Method Description**
	This method sets the project owner to userID.
	Program Description Language

	Public setProjectOwner(userID){
	  Access *projectOwner* attribute of object
	  Set *projectOwner* = string argument
	}

getProjectStatus();	**Method Description**
	This method gets a projects status, it allows for getters to be located by the projects unique projectID.
	Program Description Language

	Public getProjectStatus(){
	  Get *status* attribute of object
	  Return *status* attribute
	}

setProjectStatus(projectStatus);	**Method Description**
	This method sets a project's status to projectStatus.
	Program Description Language

	Public setProjectStatus(projectStatus){
	   Access *status* attribute of object
	   Set *status* = string argument
	}

addUserToProject(userID);	**Method Description**
	This method adds a user to a project based upon a user's unique userID.
	Program Description Language

	Public addUserProject(userID){
	  Locate User object by inputted *userID*
	  Add User object to Project users
	}

addProjectOwner(userID);	**Method Description**
	This method adds a project owner to a project based upon a user's unique userID.
	Program Description Language

	Public addProjectOwner(userID){
	  Locate User with *userID*
	  Add User to Project *projectOwner*
	}

Enumeration name: ProjectStatus	
Description: This enumeration contains a list of the options for project statuses.	
Enumerator	**Description**
Planning	This value represents a project that is in the planning and brainstorming stage.
In Development	The value represents a project that is being developed and coded.
Testing	This value represents a project that is being tested and validated
Deployment	This value represents a project that is completed with development and in the deployment stage.
Maintenance	This value represents a project that is in the maintenance stage.

Interface name: ProjectBoard	
Description: The ProjectBoard interface includes methods that allow Project objects to interact with Issue objects.	
Methods	
changeIssueStatus(issueID);	**Method Description**
	This method changes the status of an Issue object based on the issueID.
	Program Description Language

	Private changeIssueStatus(issueID){
	  Locate Issue with *issueID*
	  Call Issue's *changeStatus(IssueStatus)* method to change status
	}

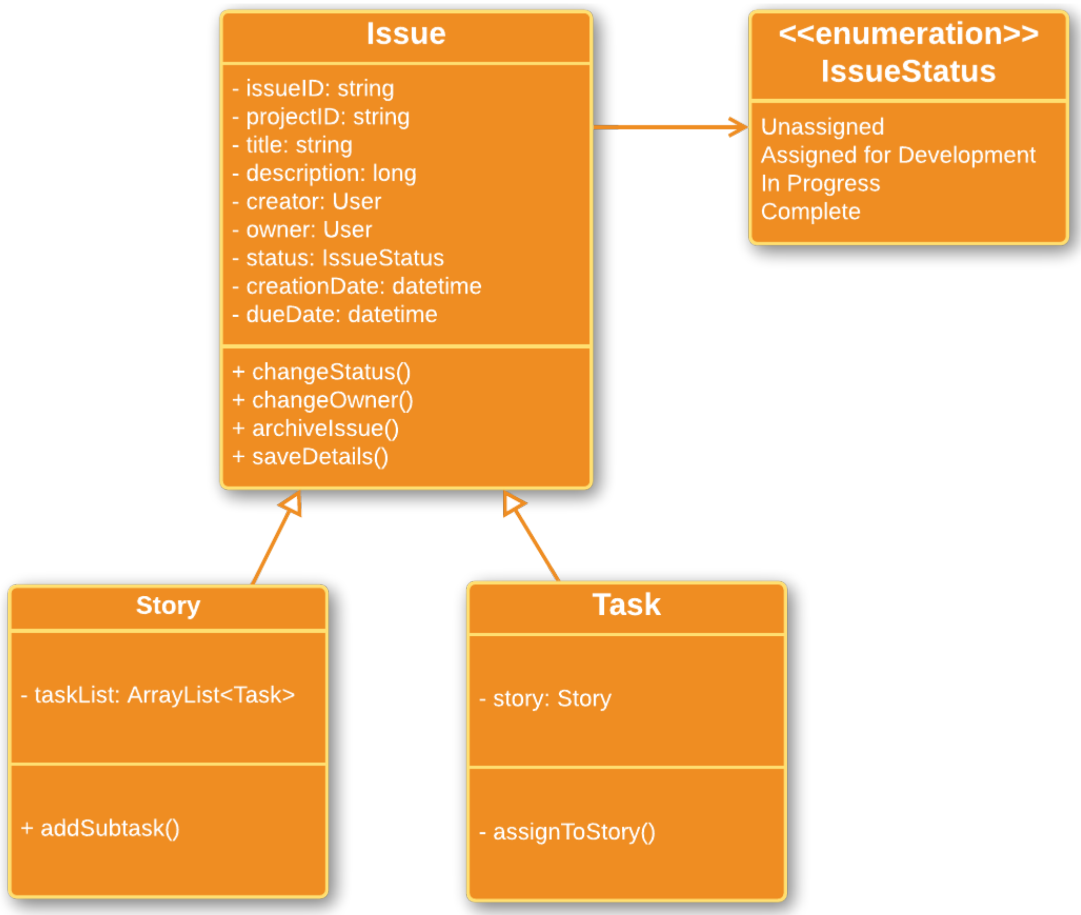
openIssueDetails(issueID);	**Method Description**
	This method opens the details of an Issue object based on the issueID.
	Program Description Language

	Private openIssueDetails(issueID){
	  Locate Issue with *issueID*
	  Display the Issue's details
	}

assignIssueToUser(issueID, userID);	**Method Description**
	This method assigns an Issue object to a user based on the userID.
	Program Description Language

	Private assignIssueToUser(issueID, userID){
	  Locate Issue with *issueID*
	  Locate User with *userID*
	  Assign the User object to the Issue
	}

2.2.4 Issue



Class name: Issue	
Description: This Class will help in identifying all information regarding issues in a project.	
Attributes	
String issueID;	**Attribute Description**
	This is a string attribute that designates a unique issueID.
	Program Description Language
	Private String issueID;

String projectID;	**Attribute Description**
	This is a string attribute that designates a unique projectID.
	Program Description Language
	Private String projectID;
String title;	**Attribute Description**
	This is a string attribute that designates a specific title of an issue.
	Program Description Language
	Private String title;
Long description;	**Attribute Description**
	This is a type long attribute that labels an issue with a description.
	Program Description Language
	Private Long description;
User creator;	**Attribute Description**
	This is an object of the user class that designates a creator of an issue.
	Program Description Language
	Private User creator;
User owner;	**Attribute Description**
	This is an object of the user class that designates an owner of an issue.
	Program Description Language
	Private User owner;

IssueStatus status	**Attribute Description**
	This is an object of the enumeration IssueStatus that designates the status of an issue.
	Program Description Language
	Private IssueStatus status
Datetime creationDate;	**Attribute Description**
	This is a Datetime attribute that designates the creation date of an issue.
	Program Description Language
	Private Datetime creationDate;
Datetime dueDate;	**Attribute Description**
	This is a Datetime attribute that designates the due date of an issue.
	Program Description Language
	Private Datetime dueDate;
Methods	
changeStatus(IssueStatus);	**Method Description**
	This method will be used with the enumeration IssueStatus to change the status of an issue.
	Program Description Language

	Public changeStatus(IssueStatus){
	   Access *status* attribute of object
	   Set *status* = IssueStatus argument
	}

changeOwner(User);	**Method Description**
	This method will be used to change the owner of a specific issue.
	Program Description Language

	Public changeOwner(){
	    Access *owner* attribute of object
	    Set *owner* = User argument
	}

archiveIssue();	**Method Description**
	This method will be used to archive an issue.
	Program Description Language

	Public archiveIssue(){
	    Change Issue object's status to Archived
	    Remove issue from board visibility
	}

saveDetails();	**Method Description**
	This method will be used to save the details of an issue.
	Program Description Language

	Public saveDetails(){
	    Get input data from Issue Detail interface
	    Store input data in respective fields of Issue
	}

Subclass name: Story	
Description: The Story subclass is a child of the Issues class. This subclass manages a specific initiative within a Project.	
Attributes	
ArrayList<Task> taskList;	**Attribute Description**
	This is an array list of Task objects that are part of the Story object.
	Program Description Language
	Private ArrayList<Task> taskList;
Methods	
addSubtask();	**Method Description**
	This method will add a Task to the story's taskList.
	Program Description Language

	Public addSubtask(){
	  Create a new Task object
	  Add new Task object to *taskList*
	}

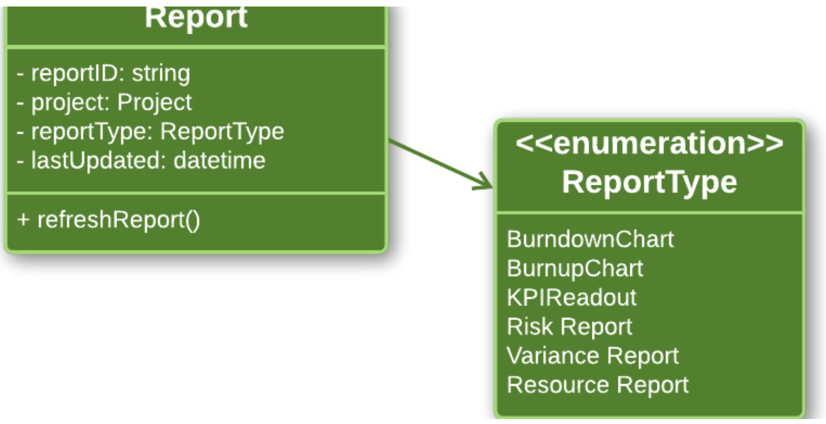
Subclass name: Task	
Description: The Task subclass is a child class of the Issue class. This subclass controls a specific goal that pertains to a Story.	
Attributes	
Story story;	**Attribute Description**
	This is an object of the Story class that the task is assigned to.
	Program Description Language
	Private Story story

Methods	
assignToStory();	**Method Description**
	This method will assign a task to a Story object.
	Program Description Language

	Private assigntoStory(){
	  Add Task object to *story*'s *taskList*
	}

Enumeration name: IssueStatus	
Description: This enumeration defines the IssueStatus data type. Each enumerator specifies the status of an issue.	
Enumerator	**Description**
Unassigned	This value represents an issue that has not been assigned.
Assigned for Development	This value represents an issue that has been assigned for development.
In Progress	This value represents an issue that is currently being worked on.
Complete	This value represents an issue that has been completed.

2.2.5 Reports



Class name: Reports	
Description: The Reports class will be responsible for maintaining report information and updating reports.	
Attributes	
String reportID;	**Attribute Description**
	This is a declaration of the report's identification.
	Program Description Language
	Private String reportID;
Project project;	**Attribute Description**
	This is a Project object that is being reported on.
	Program Description Language
	Private Project project;
ReportType reportType;	**Attribute Description**
	This is an object of the enumeration ReportType that specifies the type of report.
	Program Description Language
	Private ReportType reportType

Datetime lastUpdated;	**Attribute Description** This is a declaration of the date and time the report was last updated. **Program Description Language** Private Datetime lastUpdated;
Methods	

refreshReport();	**Method Description**
	This method will update a report.
	Program Description Language
	Public refreshReport(){
	    Access *reportType* attribute of object
	    Calculate report based on *reportType*
	    Set *lastUpdated* attribute to current date/time
	}

Enumeration name: ReportType

Description: This enumeration defines the ReportType data type. Each enumerator specifies a report type that can be generated for a project.

Enumerator	Description
BurndownChart	This value represents a burndown chart that will show the amount of work left in the project versus the time.
BurnupChart	This value represents a burnup chart that will show the amount of work completed versus the total scope of the project.
KPIReadout	This value will represent KPI readouts that will show progress and trends based on KPI’s.
RiskReport	This value will represent risk reports that will show the overall project risk and any trends pertaining to risk.

VarianceReport	This value will represent variance reports that will show the variance between the budget and what is actually spent on a project.
ResourceReport	This value will represent a resource report that will show how resources are being allocated and utilized.

2.3 Hierarchies

2.3.1 Users

The User class is a parent class and the class that defines a typical user of the Saga application. SiteAdmin is a child of the User class, which grants sitewide administrative and management permissions, such as altering or removing content. OrganizationAdmin is another child of the User class, which grants organization-specific permissions that allow the OrganizationAdmin user to manage their organization’s projects and rosters.

2.3.2 Issues

The Issue class is a parent class for two different types of issues that can be created on a project’s kanban-style issue management board. Story is a child of the Issue class and defines a single initiative involved within a project; Story objects can contain Task objects. Task is also a child of the Issue class and defines an even more specific goal or piece of work needing done within a Story.

2.4 Refinement

The needs of the Saga application require the use of refinement to break certain parent classes into multiple subclasses to capture some key difference(s) between them.

The User inheritance hierarchy is a good example of how we're using refinement to more accurately define user behaviors and permissions. All users will have the ability to manage their own account, for example. But higher levels of permissions are needed for Organization Admin users, who can manage their own account but also their organization. Even further, Site Admin users need yet higher levels of permissions to be able to manage all users and organizations application-wide. This relationship is detailed above in section **2.3.1**, with specifics outlined in the class descriptions available in **2.2.1**.

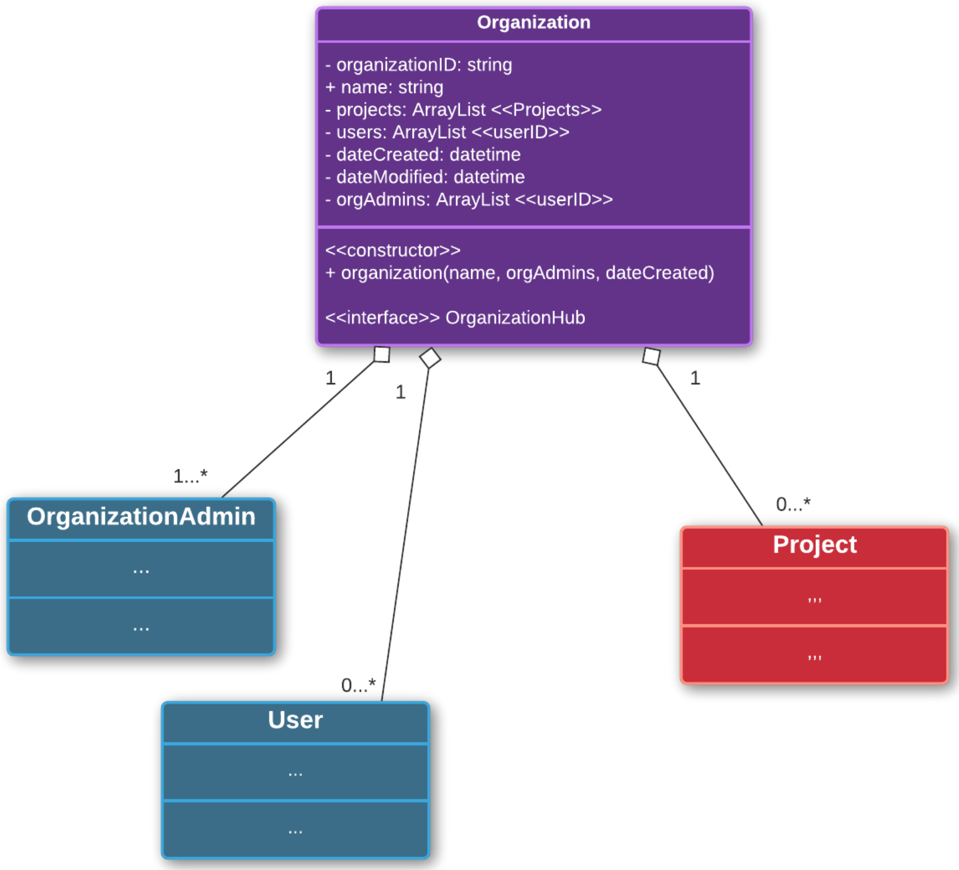
Another area where we will implement the concept of refinement is with the Issue class, which requires differentiation between types of issues that a user can create on their project's kanban-style board. Stories and Tasks both have much in common, except that a Story can contain one or more Tasks, whereas a Task cannot contain other Tasks.

2.5 Generalization

The concept of generalization is an important one for Saga to implement. The main class that will be taking advantage of generalization is the Report class. Although an Organization needs to access many types of reports with slight differences between them, they are functionally the same in how they are being utilized by the application.

2.6 Object Composition

2.6.1 Organization

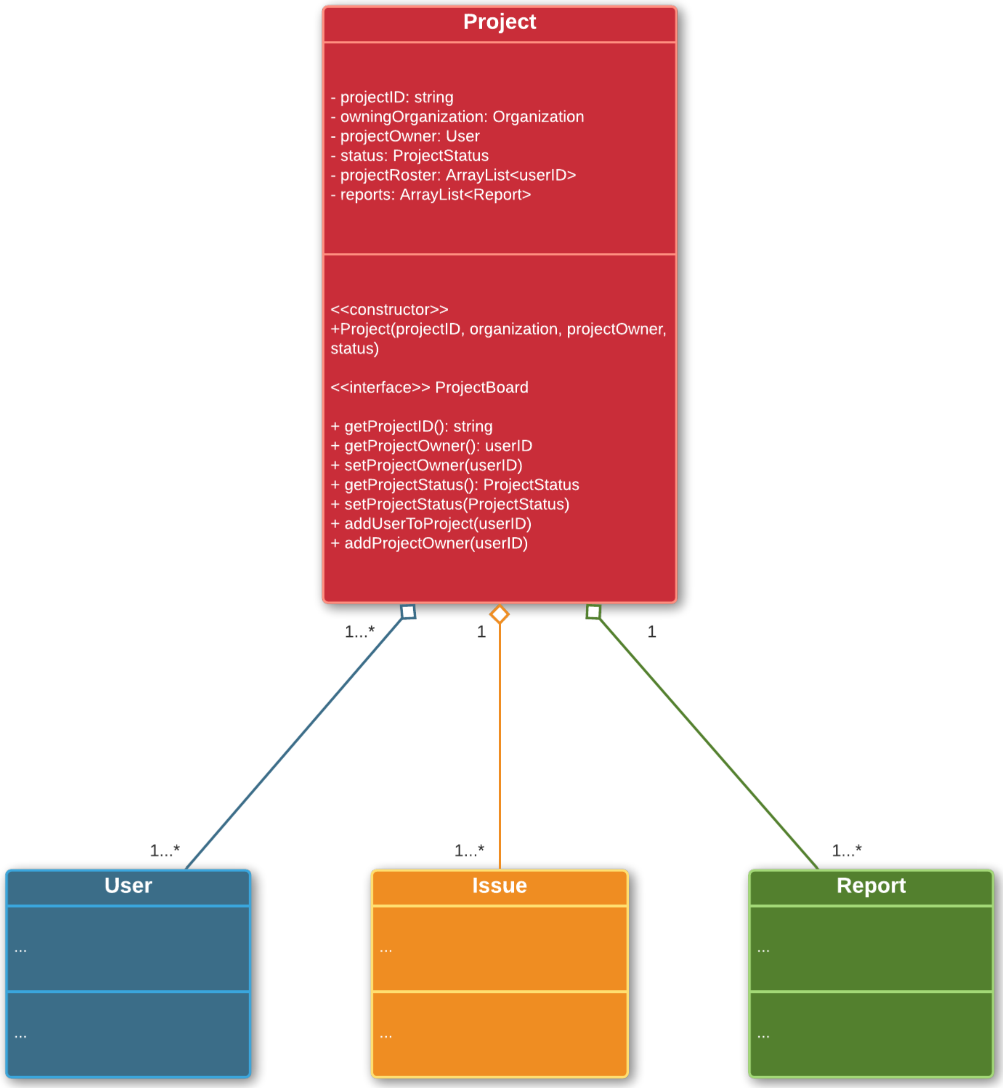


An Organization is composed of many different types of objects, and is the central connecting entity for those objects to interact with one another.

An Organization is made up of one or more different users: both User objects and OrganizationAdmin objects. These are referenced within the Organization as separate rosters that are maintained as array lists of userID's for the respective rosters. Every Organization has at least one OrganizationAdmin. Users and OrganizationAdmins can be members of multiple organizations.

An Organization is also composed of zero or more projects. These projects are referenced within the Organization as an array list of Project objects. An organization can have zero, one, or multiple Projects, but a Project can only belong to one organization.

2.6.2 Project


Project
- projectID: string
- owningOrganization: Organization
- projectOwner: User
- status: ProjectStatus
- projectRoster: ArrayList<userID>
- reports: ArrayList<Report>
<<constructor>>
+Project(projectID, organization, projectOwner, status)
<<interface>> ProjectBoard
+ getProjectID(): string
+ getProjectOwner(): userID
+ setProjectOwner(userID)
+ getProjectStatus(): ProjectStatus
+ setProjectStatus(ProjectStatus)
+ addUserToProject(userID)
+ addProjectOwner(userID)
1...*
1
1
1...*
1...*
1...*
User
Issue
Report
...

A Project is primarily composed of Issues (Story and Task types). A Project can contain zero, one, or multiple Issues, but an Issue can only belong to one Project.

A Project is also composed of Reports. A Project has multiple Reports, but a Report can only belong to a single Project.

Similar to an Organization, a Project also is composed of Users. There is an “owner” of the project, which is referenced as a User object. There is also a roster of project members, which is maintained as an array list of userIDs.

3. Database Design

3.1 Overview

3.1.1 Description of Database Usability

Saga's database is our central hub where all data will be archived and retrieved for use within various functions throughout the application. Currently, the database will hold about 7 different tables within it. Some changes have been made to accommodate the various features and functions the application should be able to support. These features include what is being displayed to the user and the ability users will have to create several different objects. The objects can include organizations, projects, issues, reports, etc. Each field within each table was made to hold specific information that is integral to the success of the application. Each field and its purpose will be defined in more detail in the upcoming sections.

3.1.2 Changes/Improvements

In almost every one of the specific entities/tables within our database, we have altered the fields and what data will be stored within the table. As we continue to dive deeper into the classes, methods and functionality of the application we found that these changes were very important to ensuring the application works with the utmost efficiency and that each functionality operates as it is intended to.

- Within the Users table we have added a field for phone number, street address, paid org member, and organization membership.
- In the Organizations table we took our userID, OrgAdmin, SiteAdmin, and added Users list and Org admins.
- In the Projects table we removed the list of issues and list of message IDs and added owning Organization, Project Owner and Status.
- We completely removed the Messages table as we agreed that this feature might be out of the scope for this application's initial version.
- Our team added a new table for Reports. This table will contain the fields Report ID, project, report type and last updated.
- Our Issues table had the most changes made to it. We decided to take out the issue name, list of stories and list of tasks. Within this table we added project, title, description, creator, owner, status, creation date, due date and type.
- We also changed both the Stories and Tasks tables.
    - For the Tasks table we took out all three original fields (task id, task name and task description), and instead we added issue ID, standalone and story.
    - For the Stories table we also took out all the four original fields (story ID, story name, story description and list of tasks), and instead we added the fields issue ID and Tasks.

Our team will go over each one of the fields within each table, in more detail, on what their purpose is in section **3.3** of the document.

3.2 Database

3.2.1 Use of Lists

Within our database, there are several fields that utilize lists to hold information in its corresponding cells. There are several features within Saga that have several different objects to display to users. For example, within every organization there can be several different projects, and within every organization there can be several users. There are many other areas where something similar would be the case throughout our application. Our team has to be able to accommodate for each project having several issues, or story having several tasks, or organization having more than one organization administrator.

After consideration on how to best approach this issue, our team thought that it might be best to utilize lists within our database which can hold all these different objects within a respective object. In many fields we used the concept of array list in order to better fit this need. This would give us many benefits in the overall functionality of the application and when writing our code. When a user wants to see which organizations they are a part of, it is easier for our code to pull that information out of the database, utilize a split function to separate each individual object and then display that to the user.

This would also help us to speed up the process of retrieving information and give the user an overall better experience when using the application. Rather than having to search several different cells within a table to find the information we need, it would be easier to only search in one cell and be able to pull that information. This would reduce the backend processing time and improve the amount of time needed for each page to load.

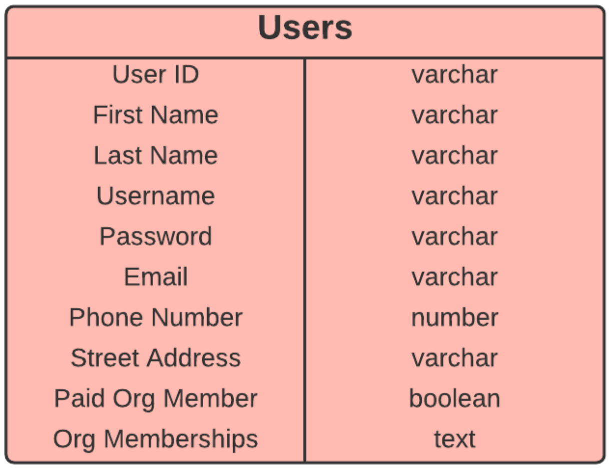
3.3 Tables

3.3.1 Users Table

The Users table will hold information for all users who currently have an account within the application. Each individual user will be given their own unique userID within the table to differentiate them and their profile information from all other users. Other fields that will be prevalent in this table will be the first name, last name, username, password, email, phone number, street address, paid subscriber, and organization membership. The Users table picture below shows the data types of each field.

For each user the password will be stored in the table as an encrypted message based on the PBKDF2 hashing function. Two important fields to highlight would be the paid subscriber field and organization memberships. The paid subscriber field will hold a boolean type value to show whether the specific user is a member of an organization that is subscribed to a paid version of the Saga app or not. This information will be valuable for determining features users can access, charging for our services, improving business functionality, etc. The other field, organization membership, will consist of an array list of unique organizationIDs which a specific user is associated with. The field can help to improve the efficiency in retrieving data to display information on our UI in order for users to view and interact with different organizations they are a part of. The purpose of all other fields within this table are straightforward in respect to the name of the field and what data each cell will hold.

Within this table, the user ID field will be our primary key. This field will uniquely identify each user object created in order to avoid any redundancy issues. The org memberships field will be the only field in this table to act as a foreign key due to the fact that it will be used to store objects from the organizations table within a list of organizations of which a user is a member.



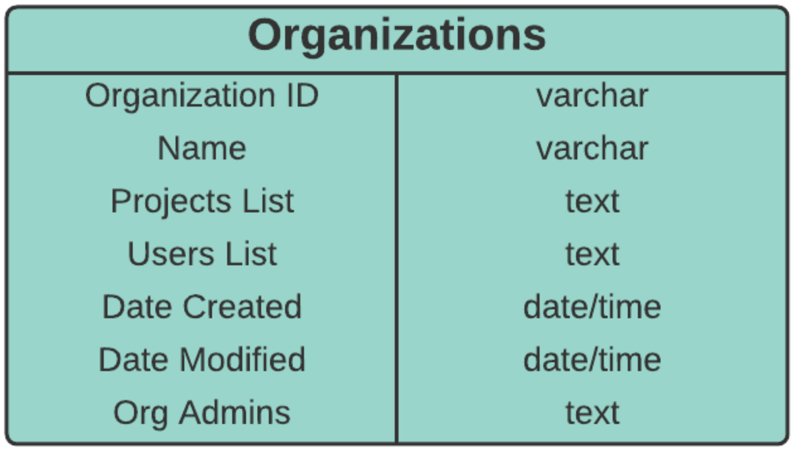
Users	
User ID	varchar
First Name	varchar
Last Name	varchar
Username	varchar
Password	varchar
Email	varchar
Phone Number	number
Street Address	varchar
Paid Org Member	boolean
Org Memberships	text

3.3.2 Organizations Table

The Organizations table will hold all the information that will be created for each individual organization. Each organization will be individually identified with a unique organizationID. This ID will be used to differentiate an organization and information pertaining to it from all specific organizations. Other fields that will be in this table include the organization name, projects, users, creation date, modification date and organization admins.

Three of these fields will consist of lists, and they include the projects, users, and organization admins fields. We decided to use lists to store data within the cells of these fields because each field can hold multiple data. Within every organization there can be more than one project, user member or organization administrator. Therefore, it is easier to retrieve data by using lists within these specific fields. Each cell in the creation date field will hold the date and time that its respective organization was created. Also, each cell within the modification date will hold the date and time its respective organization was last used or had any activity. These two go hand in hand with a feature this application will hold. The specific feature these two fields can help target is the application's ability to hide an organization from a user's main page if it has been inactive for more than one year. The organization will be stored in the database and can always be reactivated by the organization admin at a later date. Finally, the organization name field will contain the name the organization admin gives the organization. Having a name in place will help users to differentiate each individual organization, which they are a member of, on their dashboard.

The primary key of the Organizations table will be the organization ID field. This ID will help to separate each organization object from one another. This table will also have two foreign keys. The projects list and users list fields will act as the foreign keys for this table because they are using data that is retrieved from both the Projects table and Users table respectively.



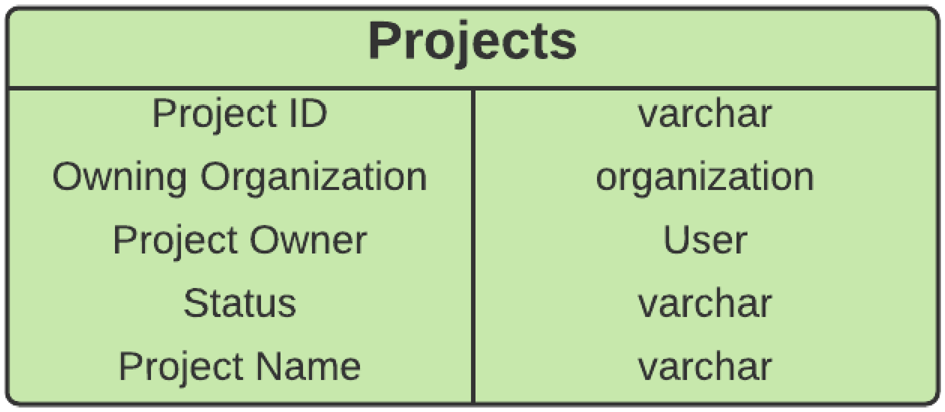
Organizations	
Organization ID	varchar
Name	varchar
Projects List	text
Users List	text
Date Created	date/time
Date Modified	date/time
Org Admins	text

3.3.3 Projects Table

The Projects table will hold information for each individual project created within the application. Each project will have its own unique project ID to identify it from other projects. This ID will be the primary key within the table. Other fields that will be included within this table will be the owning organization, project owner and status.

The owning organization field will include the specific organization that the project was created within. The organization will be identified by the organization ID. In this case, this field will be considered to be a foreign key. The project owner field will include the specific organization admin which created the project, and will be identified by their specific userID. In this way, this field will also be a foreign key. The status field will represent the overall status of the project. The values that may go into this field may include planning, development, testing, deployment, or maintenance. Finally, the name of the project, in which users gave it to identify it, will go into the project name field.

The primary key within the Projects table will be the project ID field. The project ID will give each project object a unique attribute to differentiate them from each other. This table will also have two different foreign keys. The two foreign keys within this table will be the owning organization field and the project owner field. Although data in these two fields will not be stored within lists, we are still pulling data from other tables for these two fields. The owning organization field will pull information from the Organizations table and the project owner field will pull information from the users table.



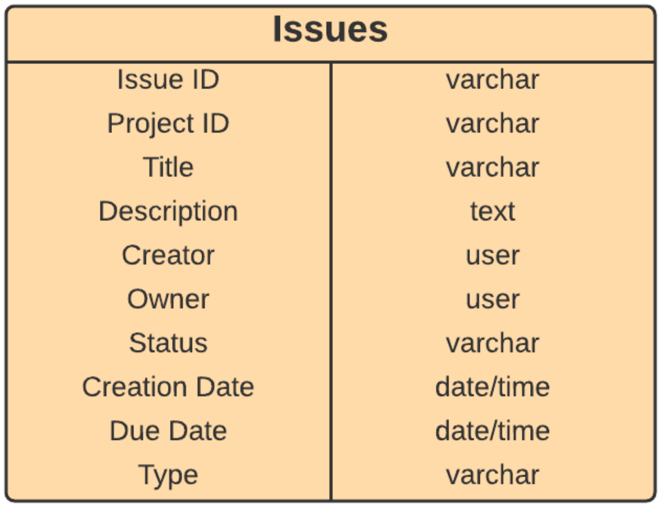
Projects	
Project ID	varchar
Owning Organization	organization
Project Owner	User
Status	varchar
Project Name	varchar

3.3.4 Issues Table

The Issues table will store information for each individual issue within each project. Every new issue created will have its unique issueID to differentiate it from all other issues. Other fields that this table will include will be projectID, title, description, creator, owner, status, creation date, and due date. The projectID will act as a foreign key in order to determine which project any specific issue was created in.

The title and description fields will include the name or phrase given to the issue and the description to define its purpose respectively. Each cell under the creator field will display the name of the user who initially created the issue. Within the owner field, each cell will display the user who is 'responsible' or taking on a specific issue. We decided to separate these two because it's possible that throughout the life of a project there may be issues where various users may switch off on who's working on it. The status field will display the phase the issue is at in terms of when it will be completed. The different statuses our application will have available for each issue are 'unassigned', 'assigned for development', 'in progress', and 'complete'. The next two fields included in this table will be creation date and due date. These fields will hold the date the issue was created and the date that team members would like to have the issue completed by respectively. Finally, the type field will hold data on whether the issue is considered to be a Story or a Task.

The primary key within the Issues table will be the issue ID. Each issue created will have its own unique ID to differentiate it from other issues. This table will have three foreign keys. The foreign keys within this table will be the project ID field, the creator field and the owner field. Both the creator and owner fields will utilize information from the users table. The project ID field will use data from the Projects table to identify which project a specific issue was created in.

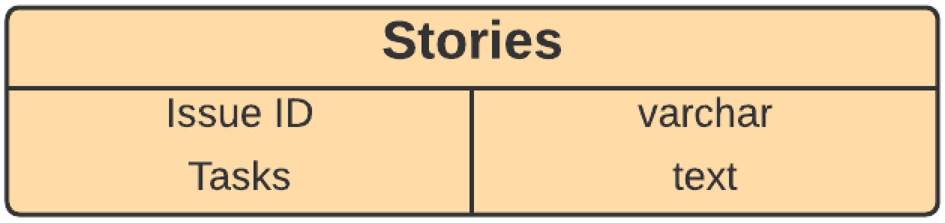


Issues	
Issue ID	varchar
Project ID	varchar
Title	varchar
Description	text
Creator	user
Owner	user
Status	varchar
Creation Date	date/time
Due Date	date/time
Type	varchar

3.3.5 Stories Table

Saga's Stories class will inherit from the Issues class. In this way they will also have their own separate issue ID. Within each story, it will be possible to have more than one task. Therefore, we have decided that this table will include a list of tasks so that our application will have a place to store the tasks that are included within each story. It is not mandatory that a story include a task within it. Due to this, a cell within this field can be null.

The Stories table will have only two fields. The issue ID field will be the primary key for this table because it is the unique attribute that will differentiate all stories from one another. The foreign key within this table will be the tasks field because it is storing data from the Tasks table.

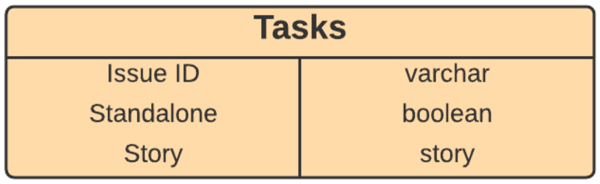


Stories	
Issue ID	varchar
Tasks	text

3.3.6 Tasks Table

The Tasks table will only consist of two fields, similar to the Stories table. The Task class will inherit from the Issues class, and due to this it will also be identified through the use of an issue ID. Our team also decided that it would be best to include a standalone field within this table as well. While a task can be made within a story, it can also be made as a standalone as well. This field will store a boolean value which will indicate whether it is standalone or not. The other field that this table will have is the story field. In the case that a task is attached to a story, then each respective cell will hold the story that the task was made within. Since every row may not have a value in the field, this will be able to have a null value if necessary.

The Tasks table will have one primary key and one foreign key. Each task created will have its own specific issue ID. This is the primary key that gives each row, or task object, a unique identifier. The story field will be the foreign key of this table as it will pull information from the Stories table.



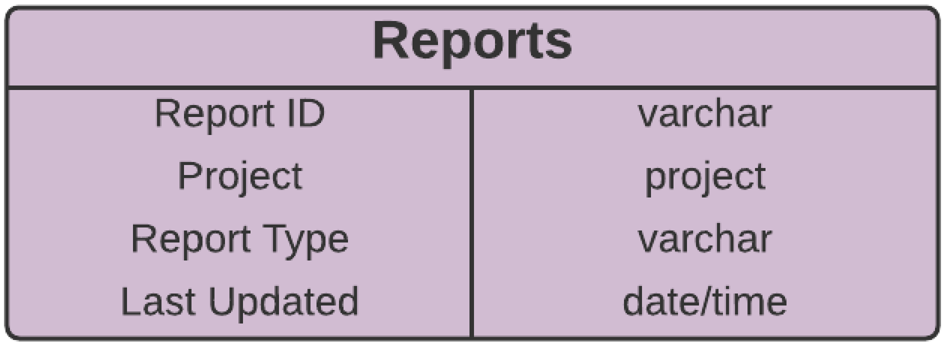
Tasks	
Issue ID	varchar
Standalone	boolean
Story	story

3.3.7 Reports Table

Our application will allow teams to create various different reports within their organization for various different projects that they are working on. Therefore, our team thought it would be necessary to have a table that could hold data for each report to keep better track of each one and ensure proper functionality within the application. Each report will be assigned its own reportID in order to differentiate it from other reports. Other fields that will be included within the table will include project, report type, and date last updated.

The project field will include the project ID of the project which the report was created within. This ensures that we can better track each report, and in which project it was created within in case team members need to retrieve the report for any reason. The report type field will show what type of the report corresponds with its respective reportID. Within our application we decided to allow users to create various different reports to aid in viewing their overall progress. The reports they will be able to create are Burndown Charts, Burnup Charts, KPI Readouts, Risk reports, Variance reports, and Resource reports. Depending on the type of report the team selected, one of these will go into the cell, in respect to its reportID, under the report type field. Finally, the date last updated field will include a date/time of when any specific report was last updated by any team member.

Each report created will have its own report ID as a primary key to uniquely identify it among other reports stored in the table. The project field will act as the foreign key in this table as it will take information from the Projects table to store which specific project a report was created in.

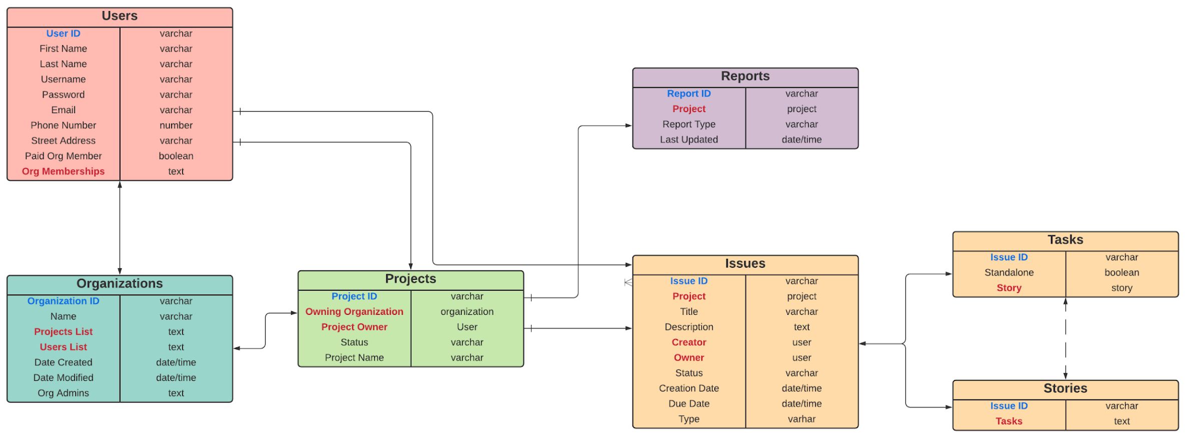


Reports	
Report ID	varchar
Project	project
Report Type	varchar
Last Updated	date/time

3.4 Relationships

3.4.1 Primary and Foreign Key Relationships

The following diagram depicts an entity relationship diagram that shows the relationships between the different tables that will be included within Saga's database. All primary key fields have been colored in blue. Each table has a limit of one primary key in order to help differentiate each row to relate to a unique object. This will benefit the functionality of the application, and reduce confusion on relations between tables. Each table also has at least one foreign key. Every foreign key will be in bold and highlighted in red within each table. Foreign keys will help show how each table is interacting with other tables. Within our application, different features will utilize several different objects. Examples of this were given in the above section **3.2.1**. These foreign keys also help to show how features are utilizing these objects from a database perspective.



3.4.2 One-to-One Relationship

Within our database there are no One-to-One relationships between our tables.

3.4.3 One-to-Many Relationship

Projects to Organizations/Reports/Issues:

The Projects table is connected to three different tables that rely on it for information. The Organizations table pulls data from the Projects table to use in its projects list field. Each issue is connected to a specific project which is shown within the project field. This is also true for the Reports Table. Each report is connected to its corresponding project it was created in through its project field. In these ways all three tables are dependent specifically on the projects table.

Users to Organizations/Projects/Issues:

The Users table connects to several different tables at once. The Organizations, Projects, and Issues tables all depend on the Users table for the data that is held within some of their fields. Within the Organizations table, the application utilizes information from the Users table with the Users Lists field. This field holds information of all users that are currently a member of a respective organization. The Projects table depends on the Users table because it utilizes data within its project owner field, which is retrieved from the Users table. Similarly, the Issues table utilizes data from the Users table within its creator and owner fields.

Organizations to Users/Projects:

Both the Users and Projects tables depend on the Organizations table for data to utilize within their fields. The Users table uses the Organizations table for its organization membership field This field holds all the organizations that a user may be a part of, so it depends heavily on the Organizations table to have that information within it. Also, within the Projects table, the owning organization field depends on the Organizations table in order to store which organization a project object was created in. These two dependencies create a one to many relationship between the Organizations table and the Projects and Users table.

3.4.4 Many-to-Many Relationship

Issues/Stories/Tasks:

The Issues, Stories, and Tasks tables will have a many to many relationship because all of them are interacting with each other simultaneously. Each issue can be considered to be either a Story or Task, each story can hold tasks within it and each task can be either attributed to a story or be standalone. While a story and task may seem as independent when viewed, they are actually identified by an issue ID. Each issue ID is attributed to a unique issue object. Each issue object will contain all features from within the Issue table. However, due to how our application functions it is still important to have separate tables to differentiate between tasks and stories. Our team feels that it is important, therefore, to show how these three will interact with each other. A visual diagram of them can be viewed within section **3.4.1**.

4. Verification and Validation Testing

4.1 Overview

Saga is a project management application that will allow its users to create a product using multiple applications to achieve their goals. The verification process is the process of examining the system requirements throughout. This can also examine and check for any errors. This product is developed correctly whereas this will provide verification. It also confirms whether the product developed meets the requirements we have. Validation is the process of evaluating the product to check whether the software meets customers ' expectations and is likewise in the static testing process.

4.2 Test Types

4.2.1 Unit Testing

Unit testing will test the smallest testable portion of the system or application that can be compiled, loaded, and executed. This test will help us test each module individually.

The main purpose is to test each part of the software by separating it, and we can be sure that the components fulfill the functions.

4.2.2 Component Testing

The component test plays an important role in finding the problem. Before proceeding with the integration test, we always conduct component testing to make sure that each application module is working properly and according to the requirements. When we detect flaws and events without official recording, they are corrected as soon as possible.

We use debugging tools or unit test structure tools for such tests, we do it on the code written, and with the support of the integrated development environment.

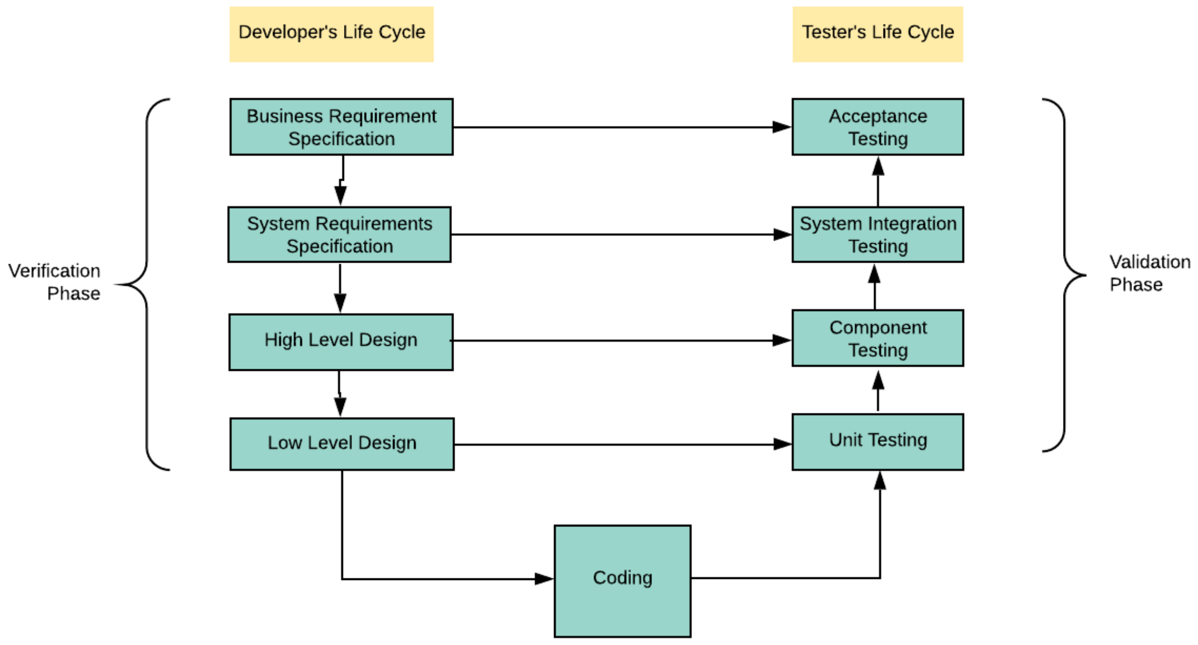
4.2.3 System & Integration Testing

During the integration testing phase, different software modules are put together and tested as a group to ensure that the integrated system is ready for system testing. The data flow from one module to the other is controlled by testers.

After the data flow from one module to another is checked by the testers, the suitability of the system is checked according to the requirements, and the overall interaction of the components is tested. A complete and integrated system test completes load, performance, reliability, and safety tests.

4.2.4 Acceptance Testing

The acceptance test is the most important stage of the test as it decides whether the customer approves the application/software. It includes the application's functionality, usability, performance, and performance. The acceptance test is a test process we perform to determine application/software needs and business processes; is a type of test done by users, customers, or other authorized organizations. Acceptance tests are the process of black box system testing. System users test according to what's going on in real-time scenarios, and it helps us check if the application meets all the technical specifications.



4.3 Test Suites and Cycles

4.3.1 Issue Creation

Test Case ID	Test Case Description
TC-0012	Verify if the user is able to click and create a new project.
TC-0013	Verify if the user is able to create a new story on existing projects.
TC-0014	Verify a user is able to create a task on existing projects.
TC-0029	Verify the user can set the due date for an issue of any type that they are creating.
TC-0030	Verify the user can set the title for an issue of any type they are creating.
TC-0031	Verify the user can set a description for an issue of any type that they are creating.
TC-0032	Verify a user can set one or more tags for an issue of any type that they are creating.
TC-0033	Verify the user can assign themselves or another member of their organization as the primary owner of an issue of any kind they are creating
TC-0034	Verify the user can set the story to which a task belongs when creating a task.
TC-0035	Verify the user can set the project to which a story belongs when creating a story.

4.3.2 Issue Management

Test Case ID	Test Case Description
TC-0025	Verify a user can change the status of a story by drag-and-dropping its card from one column to another.
TC-0026	Verify a user can assign themselves as primary owner of an issue that is not currently assigned to them.
TC-0027	Verify a user can double click on an issue's card to view the details of the issue.
TC-0028	Verify a user can modify any field or attribute of an issue within its detail view and save the changes.
TC-0036	Verify the user can add a comment to an issue of any type within its detail view and save the changes.
TC-0038	Verify the user can change the status of a story or task within the details view of that issue.
TC-0041	Verify admin users can archive issues.
TC-0042	Verify the application timestamps any major changes to an issue.
TC-0054	Verify users can select a color to label issues.

4.3.3 Organization Management

Test Case ID	Test Case Description
TC-0018	Verify a user is assigned as Admin when creating a new organization.
TC-0019	Verify a user is able to join an organization which they have been invited to.
TC-0020	Verify a user who is assigned admin can invite other users to join their organization
TC-0021	Verify a user who is admin can remove members from their organization.
TC-0022	Verify a user can remove themselves from an organization.
TC-0037	Verify the user can be a member of multiple organizations.
TC-0040	Verify users can save files based on degree of importance.
TC-0043	Verify admin users can invite new users with a copied link.
TC-0044	Verify users can search for existing groups.
TC-0064	Verify admins can invite new users with an email invitation.

4.3.4 Reports

Test Case ID	Test Case Description
TC-0048	Verify users can view analytical reports about their projects.
TC-0049	Verify users can view report data about the project's average difference between estimated task completion time and actual task completion time.
TC-0050	Verify users can view report data about their projects overall completion percentage compared to due date.

4.3.5 Security

Test Case ID	Test Case Description
TC-0051	Verify users can not view any projects, stories, or tasks that are owned by an organization of which they are not a member.
TC-0052	Verify a user can recover their account via a password reset email sent to their verified email address.
TC-0057	Verify users are receiving emails regarding new devices attempting to log into their account.
TC-0060	Verify user accounts are locked if the wrong password is entered more than five times.
TC-0077	Verify that a newly-created user's password is encrypted in the site admin tools.
TC-0078	Verify that logging into a new device causes the application to notify the user by email.
TC-0084	Verify that changes to a user account are logged with the correct timestamp in the site admin tool.

4.3.6 Service Types

Test Case ID	Test Case Description
TC-0068	Verify that advertisements are displayed when logged into an account that is not a member of a paid-service organization.
TC-0069	Verify that advertisements are not displayed when logged into an account that is a member of a paid-service organization.
TC-0070	Verify that an organization admin can purchase premium subscription with a valid credit card.
TC-0071	Verify that an organization admin can purchase premium subscription with a valid debit card.
TC-0072	Verify that an organization admin can purchase premium subscription with a valid Paypal account.
TC-0073	Verify that an organization that is not paying for premium service can have a maximum of five projects managed.
TC-0074	Verify that an organization that is paying for premium service can have unlimited number of projects managed.

4.3.7 User Identification

Test Case ID	Test Case Description
TC-0004	Verify if a user can enter their username.
TC-0005	Verify if a user can enter their password.
TC-0007	Verify if a user is logged in when valid email and password is entered.
TC-0015	Verify a user is able to log in using their unique username or email address
TC-0017	Verify a user email cannot be used to make multiple accounts.
TC-0065	Verify a new user can enter an email when creating a new account.
TC-0066	Verify a new user can enter a password when creating a new account.
TC-0067	Verify once a user has entered the proper information, a new user can create a new account.

4.3.8 User Interface

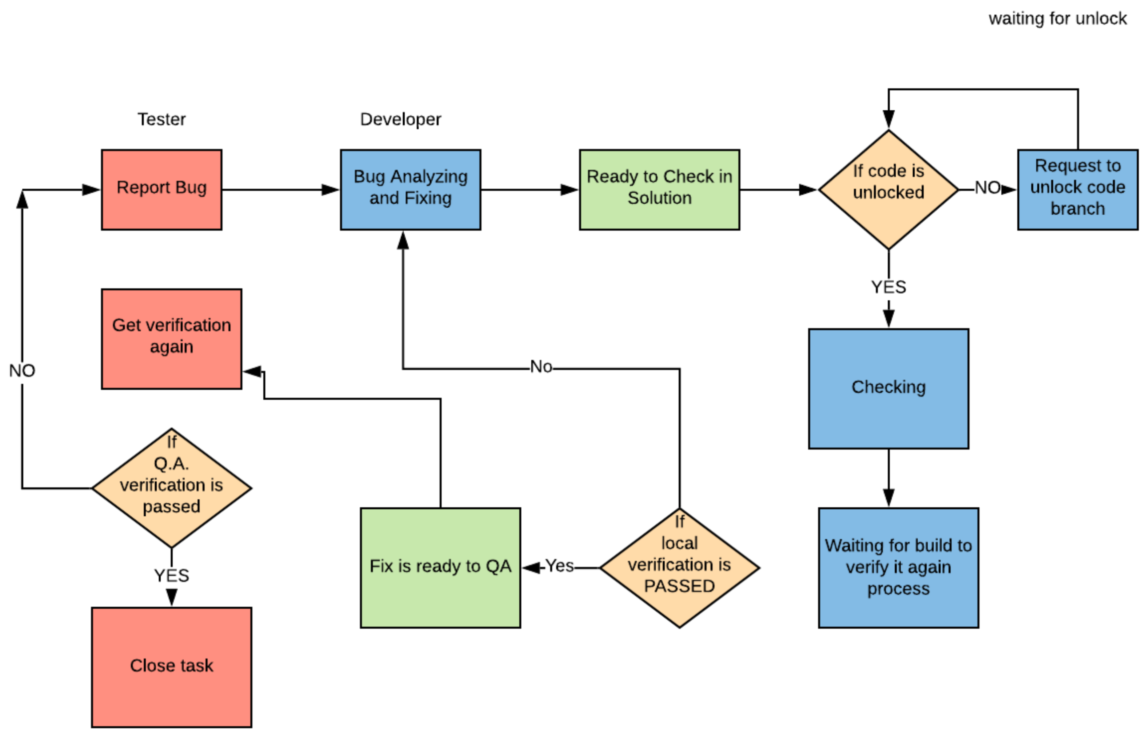
Test Case ID	Test Case Description
TC-0002	Verify if the user is able to see the home page.
TC-0003	Verify if the user is able to click on button Sign in.
TC-0006	Verify if the user can click the submit button .
TC-0009	Verify if the user is able to click to add new button.
TC-0010	Verify if the user is able to see "new contact" screen.
TC-0011	Verify if the user enter the details.
TC-0023	Verify a user with access to a project can click on that project's name to open it.
TC-0024	Verify a user can open a project's Kanban Board by clicking on the Kanban Board option.
TC-0055	Verify the application is displaying dates based upon the format defined in the user's web browser.

4.3.9 User Profile

Test Case ID	Test Case Description
TC-0008	Verify if a user is able to see their user profile page.
TC-0046	Verify users can search for other users by username.
TC-0047	Verify users can modify all fields of their own user profile.
TC-0063	Verify users can search for other users by email address.

4.4 Defect Management

When reporting the error to the developer, our error report contains Defect_ID, Defect Description, and Detected By information. Defect Management goes through a systematic process to identify and correct errors, defect closure, defect reports at the end of the project. During the discovery phase, our project team should find as many defects as possible before being discovered by the end Customer. It is recorded that a defect has been discovered and changed to accepted status when the defect is received by the developers. Classification of defects helps developers to eliminate defects that are very important first.Once defects are accepted and categorized, the process proceeds as an assignment, schedule fixing, fix the defect, and reporting the solution.


waiting for unlock
Tester
Developer
Report Bug
Bug Analyzing and Fixing
Ready to Check in Solution
If code is unlocked
NO
Request to unlock code branch
YES
Checking
Waiting for build to verify it again process
Get verification again
No
NO
If Q.A. verification is passed
YES
Close task
Fix is ready to QA
Yes
If local verification is PASSED

4.5 Testing Phases

4.5.1 Functional Testing

During the functional test, the definition of the functions that the software should perform, data input and entry, execution of the test case, and analysis of the actual results are performed. During a functional testing suite real system usage is simulated to get as close as possible to real system usage and to create test conditions related to user requirements.

4.5.2 Usability Testing

Usability aims to combine functionality testing and the overall user experience. The goals of usability testing are to ensure that the application helps users achieve their goals quickly and easily, to uncover confusion areas, and to improve the overall user experience.

4.5.3 Interface Testing

Interface testing ensures that all interactions between the web server and application server interfaces run smoothly, ensuring that error messages are displayed correctly, as well as checking these communication processes. It also helps to verify security requirements as communication spreads across systems.

4.5.4 Compatibility Testing

Testing web applications will be an important step to ensure that our Saga application is compatible with all browsers and devices.

4.5.4.1 Browser Compatibility

Browser compatibility testing contains a working environment, browser notifications, and authentication requests designed to ensure that our application works properly on different browsers. It also helps us check if any updates affect its functionality.

4.5.4.2 Operating System Compatibility

To avoid problems on some operating systems, we shall test the application to run on different operating systems (e.g. Windows, macOS, Linux, and Unix).

4.5.4.3 Mobile Compatibility

Mobile compatibility testing will ensure that the application runs according to requirements and standards on the web browser of different devices, including Android and iOS devices.

4.5.5 Performance Testing

After making sure that the functionality of the application works properly and responsibly on all browsers and devices, it will be necessary to test how it works under a heavy usage load. It will be necessary to test the application at different internet speeds (load test) and to determine the breakpoint of the application (stress test). Performance testing is a crucial test to find out how the application works under stress before users place that stress on it.

4.5.6 Security Testing

The last step, the security test, ensures that your application is protected against viruses or other malicious actions: safety testing for the application; testing whether secure pages can be accessed without authorization; ensuring that restricted files cannot be downloaded without proper authorization; and ensuring that sessions remaining open after user inactivity will be closed.

A safety test checklist includes tasks in areas such as:

- Secure Transmission
- Authentication
- Session Management
- Competency
- Data Validation

5. Appendix

5.1 Collaboration Tools

5.1.1 Slack

This platform is a tool used for essential communication between a team or group. Saga has utilized this tool as the main form of communication through-out design and development. This tool has many API's involved, which has greatly increased the progress within the program itself. Some of the API's involved include file sharing which has been extremely vital along the SDLC of the project. We will continue to use Slack as we near the deployment stage as well as the maintenance stage.

5.2.2 Lucidchart

Lucidchart is a workspace that combines diagramming, data visualization, and collaboration to allow users to work together on items such as UML diagrams, flowcharts, and basic diagrams. Saga has used this in many aspects to gather information and visualize it for our clients. We take these charts and diagrams and use them as key visuals in all of our documents. Adding flowcharts and diagrams helps our clients and consumers understand the product better by understanding what is happening and why.

5.2.3 Google G Suite

The Google G Suite consists of a variety of document collaboration tools such as Google Docs, Google Sheets, Google Slides, and Google Drive. Saga has used these tools to file share our deliverable documents including the Concept Pitch, Software Requirements, HLD and LLD. Saga has also utilized Google Sheets and Google Drive to gather information and brainstorm for our main documents. We will continue using this suite of tools for development documentation, collaboration, and creation of user documentation.

5.2.4 Google Meet/Google Hangouts

Google Meet is a communication platform that is used for teleconferencing. As a team we have scheduled regular meetings to check for proficiency and progress within our project and documentation. Google Hangouts is a related platform that allows for instant messaging between team members.

5.2.5 Trello

Trello is a kanban-style list-making application for keeping track of progress of assignments or events. Saga has used this web application to track progress and important upcoming dates or key items for the project.

5.2.6 Adobe Creative Cloud

Adobe Creative Cloud will be utilized for both design mockups and all graphic design required for the development of this application. Other Adobe CC applications may need to be used for smaller, more niche tasks, such as InDesign for project documentation needs or Illustrator for vector graphics or Dreamweaver for quick UI element mockups.

5.2.7 Jira

Jira is a project management application that allows for team communication and tracking issues, among other functions. Saga will use Jira during development to track bugs and execute its test cycles. Once Saga has been successfully deployed, this functionality will be transferred over to the Saga application itself.

5.2.8 Github

Github is a version control platform that allows teams to collaborate on projects, including functionality for branch management such as pull requests and merging branches. We will use Github as our central repository for all project-based files in order to prevent developers from working over top of one another and to facilitate collaboration.