UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

PROJECT STORM
TEAM NAME

Renaldo van Dyk 12204359
Sean Hill 12221458
Shaun Meintjes 133310896
Johann Dian Marx 12105202

Client:

Linda Marshall & Vreda Pieterse
lmarshall@cs.up.ac.za,vpieterse@cs.up.ac.za

GitHub link

Contents

1 Vision and Scope

1.1 History and Background

1.1.1 A Brief History of Project STORM

In 2013 lecturers of the department of Computer Science, which resides at the University of Pretoria, approached honors students of the module "Educational Software Development" to develop a team shuffling system. The system was to be used by the lecturers of the module "Software Engineering" to determine teams for the "Rocking the boat" exercise of the "Software Engineering" module using a set of lecturer defined criteria to create the teams with. An incomplete requirements specification document was designed and the project was halted.

1.1.2 Project Background

The lecturers of the "Software Engineering" module sought the need for such a shuffling tool, this time approaching students of the "Software Development" module. The requirements specification previously developed will be used as a starting point. This document will be stripped down to a "basic system" requirements specification, not completely discarding functionality specified in the previous documentation but adding relevant functionality as the development life-cycle persists.

1.2 Project Scope

The complete system should enable users to build teams, from a list of subjects, by selecting a set of criteria. This will aid the users in such a way that the users do not have to build the teams manually, which may require a lot of time and effort. The users can spend their time rather on analyzing the results of each "Rocking the Boat" round to change the criteria for the next round more effectively.

2 Application requirements and design

2.1 Modular Design

The system is to be a modular system which allows for:

- Only a subset of modules to be deployed. Minimally the system will require the core modules to be deployed.
- Further modules to be added at a later stage.

To this end there should be:

- Minimal dependencies between modules, and
- No dependencies of core modules on any add-on modules.

Modular design allows that each module encapsulates information that is not available to the rest of the program. This information hiding reduces the cost of subsequent design changes when future functionality is added to STORM. For example, if at a later stage functionality is added to allow for personality tests to be completed within STORM and results are automatically pulled in, a new module can be added without affecting other modules.

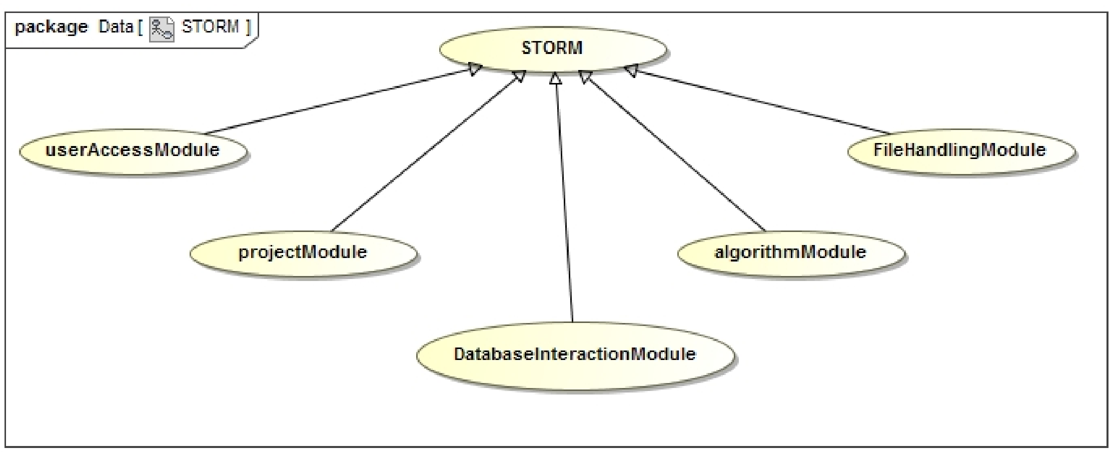

Figure 1: High level overview of STORM

2.2 User Access Module

This module deals with the STORM user access, specifically signing up, logging in, logging out and user authentication.

2.2.1 Use-cases

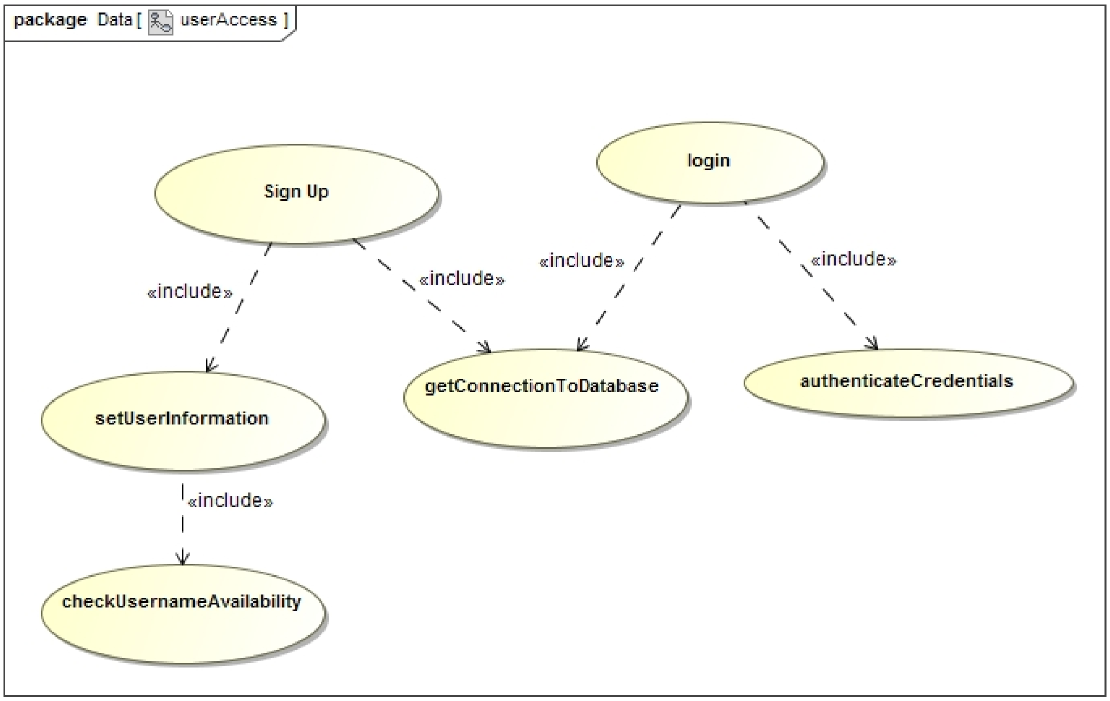

Figure 2: User access module use case

1. Sign Up

   Priority: Critical.

   Pre-condition: Client must have a valid e-mail address.

   Post-condition: Client has a STORM profile.

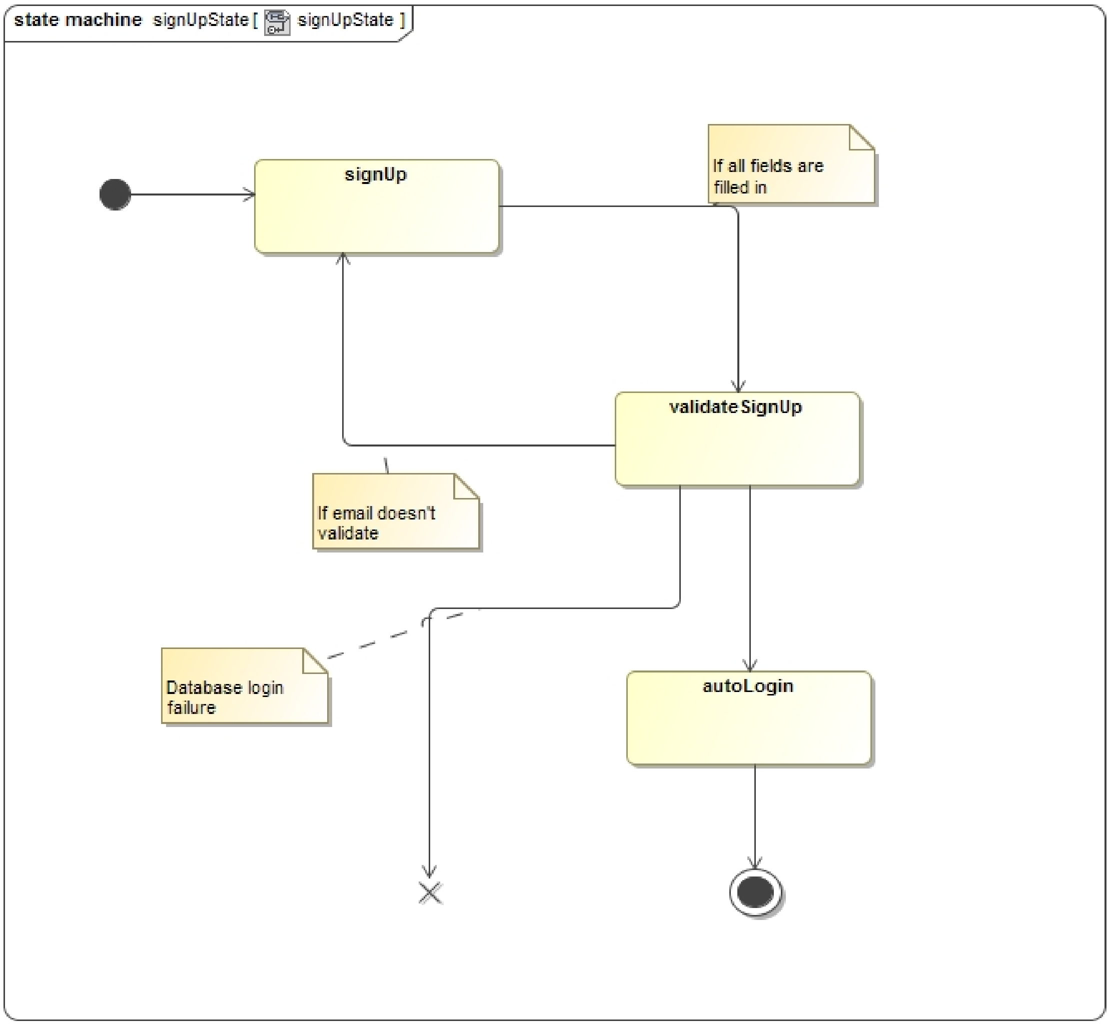

Figure 3: SignUp state diagram

2. Login

   Priority: Critical.

   Pre-condition: Client must have a STORM profile.

   Post-condition: Client can now use STORM functionality.

3. Log out

   Priority: Critical.

Pre-condition: Client must be logged into STORM.

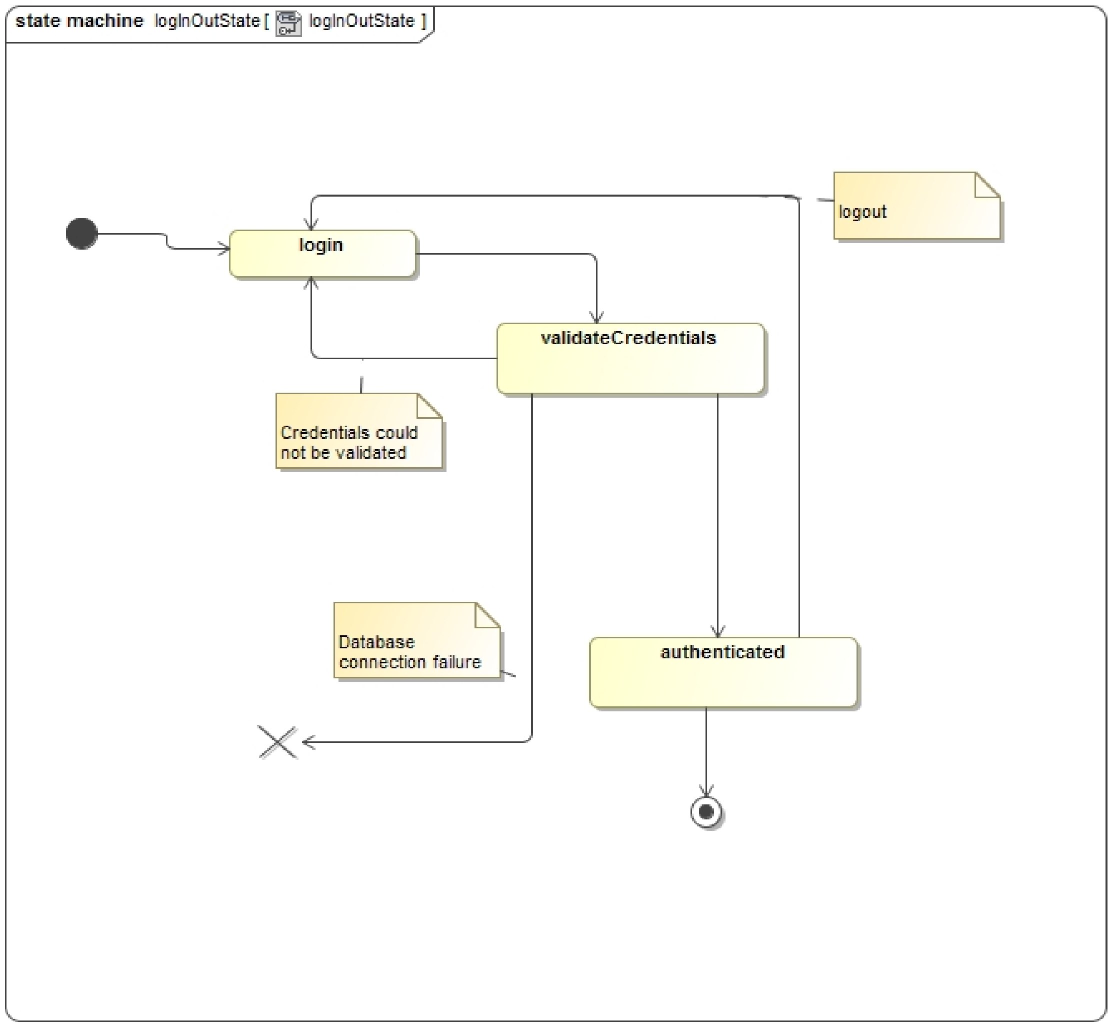

Figure 4: Login/logout state diagram

2.3 Project Module

This module deals with the configuration of projects. Initially a skeleton project will be set up with basic criteria and additional criteria can be added at a later stage as needed.

2.3.1 Use-cases

The project module provides services to create and manipulate projects.

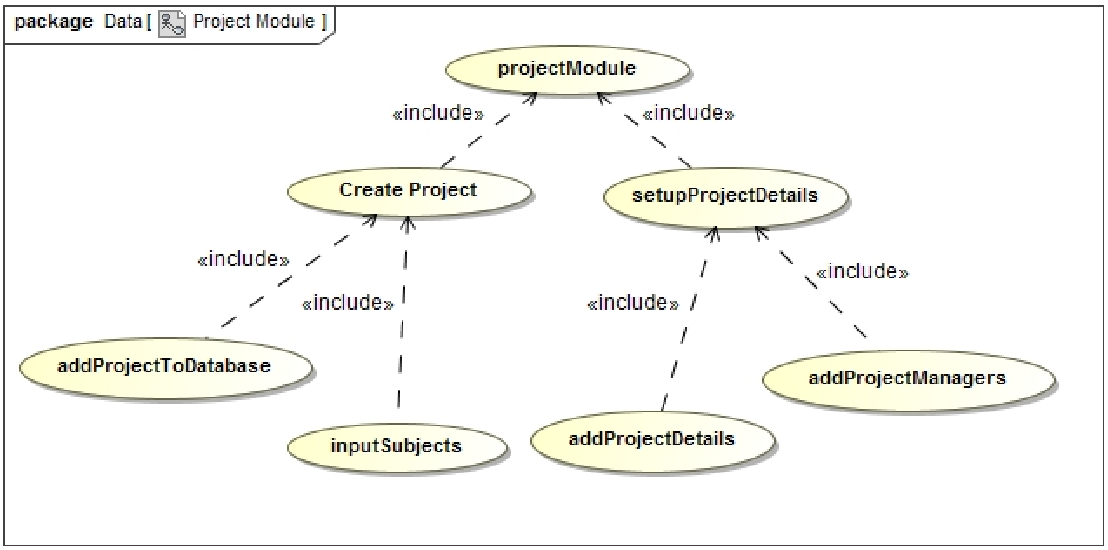

Figure 5: Project Module Use Case

1. addProjectDetails
   Priority: Critical.
   Pre-condition: Client must be logged in.
   Post-condition: Client must have a STORM profile.
   Post-condition: Project skeleton is created in database.
2. addProjectManagers
   Priority: Important.
   Pre-condition: Client must have a STORM profile.
   Pre-condition: Manager to be added must have a STORM profile.
   Pre-condition: Client must have created a STORM project.
   Post-condition: Manager has permission to collaborate on the project.

3. inputSubjects

   Priority: Critical.

   Pre-condition: Client must have a STORM profile.

   Pre-condition: Client must have created a STORM project skeleton.

   Pre-condition: A .csv file should exist and should be selected.

   Pre-condition: The .csv should have subjects in it.

   Post-condition: Project database is updated with a list of subjects.

4. addProjectToDatabase

   Priority: Critical.

   Pre-condition: Client must have a STORM profile and must be logged in.

   Pre-condition: A project should exist and should have subjects that was added by the .csv file.

   Pre-condition: The project should have a unique name that is not the same as another project for the user.

   Post-condition: Project is added to the database with the subjects and criteria.

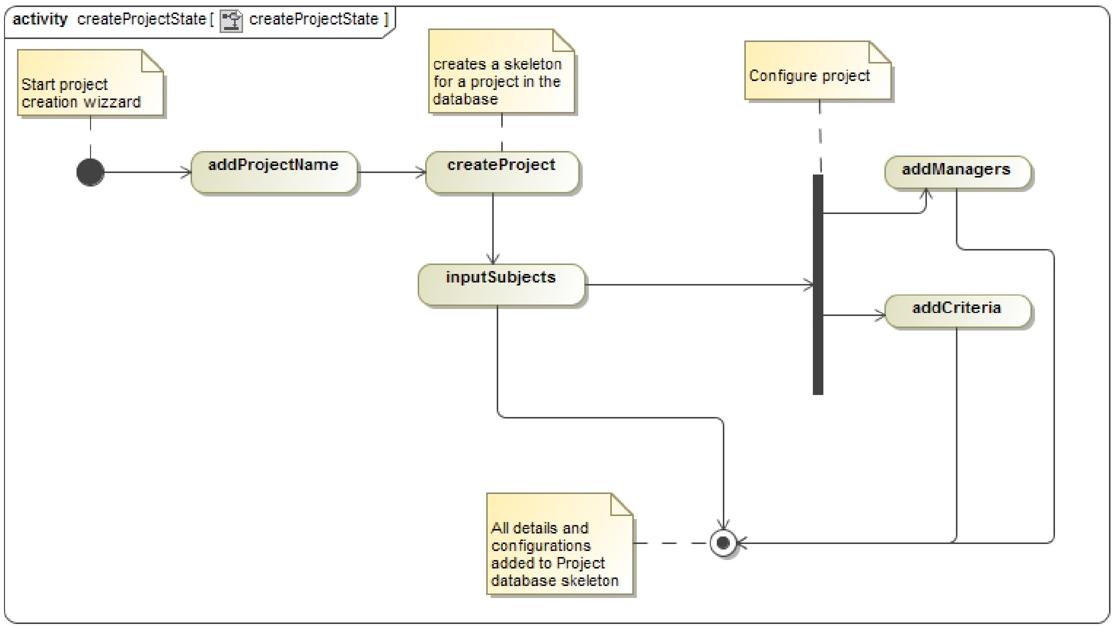

Figure 6: Create project state diagram

5. uploadCSVToUpdateSubjects

   Priority: Critical.

   Pre-condition: Client must have a STORM profile and must be logged in.

   Pre-condition: A project should exist and should have subjects that was added by the .csv file.

   Pre-condition: The .csv should be in the correct format as can be seen in the User Manual, when uploading.

   Post-condition: Updated Criteria and Subjects are added to the project.

6. EditSubjectsIndividually

   Priority: Important.

   Pre-condition: Client must have a STORM profile.

   Pre-condition: Client must be logged in and authorized.

   Pre-condition: A project should be selected.

Pre-condition: The criteria should be updated with valid values.

Post-condition: The subjects in the project is updated individually.

7. AddSubjectsIndividually

   Priority: Important.

   Pre-condition: Client must have a STORM profile.

   Pre-condition: Client must be logged in and authorized.

   Pre-condition: A project should be selected.

   Pre-condition: The subject should be added with valid values and criteria.

   Post-condition: The subject is added to the project.

8. RemoveSubjectsIndividually

   Priority: Critical.

   Pre-condition: Client must have a STORM profile.

   Pre-condition: Client must be logged in and authorized.

   Pre-condition: A project should be selected.

   Pre-condition: The subject should be selected and deleted.

   Post-condition: The subject is removed from the project.

2.4 Database Interaction Module

This module deals with the creation and interaction with the database.

2.4.1 Use-cases

After a project has been created, a database collection is created and associated with it. The project can then request and update the database and persist as the shuffling criteria grows.

1. createDB

   Priority: Critical.

   Pre-condition: The databse should not exist yet.

   Pre-condition: Client must have a STORM profile.

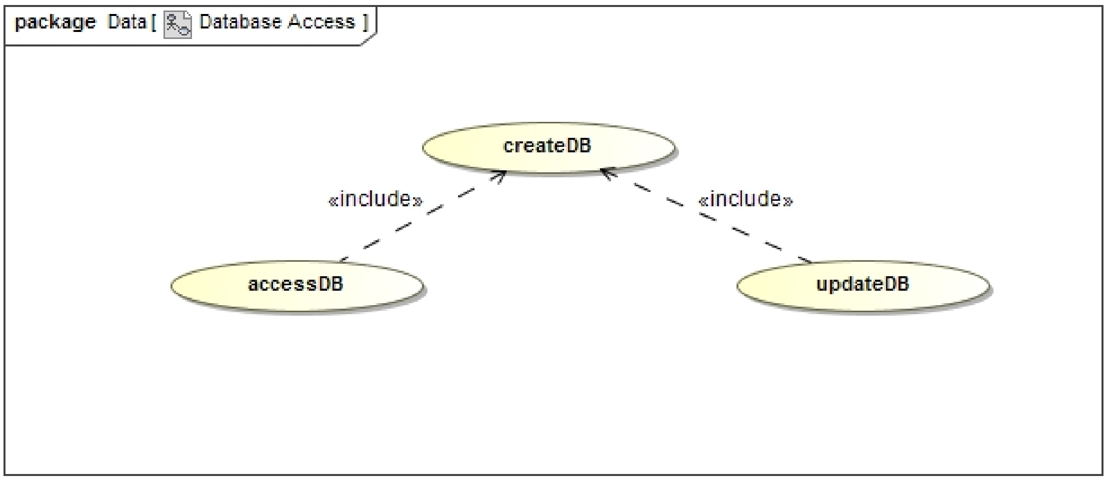

Figure 7: Database access use case

Post-condition: The database is created.

2. accessDB

   Priority: Critical.

   Pre-condition: The database should exist.

   Pre-condition: The user should have a STORM profile.

   Pre-condition: The user should be authorized to access the database.

   Post-condition: The database is accessed and queried.

3. updateDB

   Priority: Critical.

   Pre-condition: The database should exist.

   Pre-condition: The user should have a STORM profile.

   Pre-condition: The user should be authorized to update the database.

   Post-condition: The database is updated.

2.5 Algorithm Module

This module deals with the main process behind STORM, it is the shuffling algorithm.

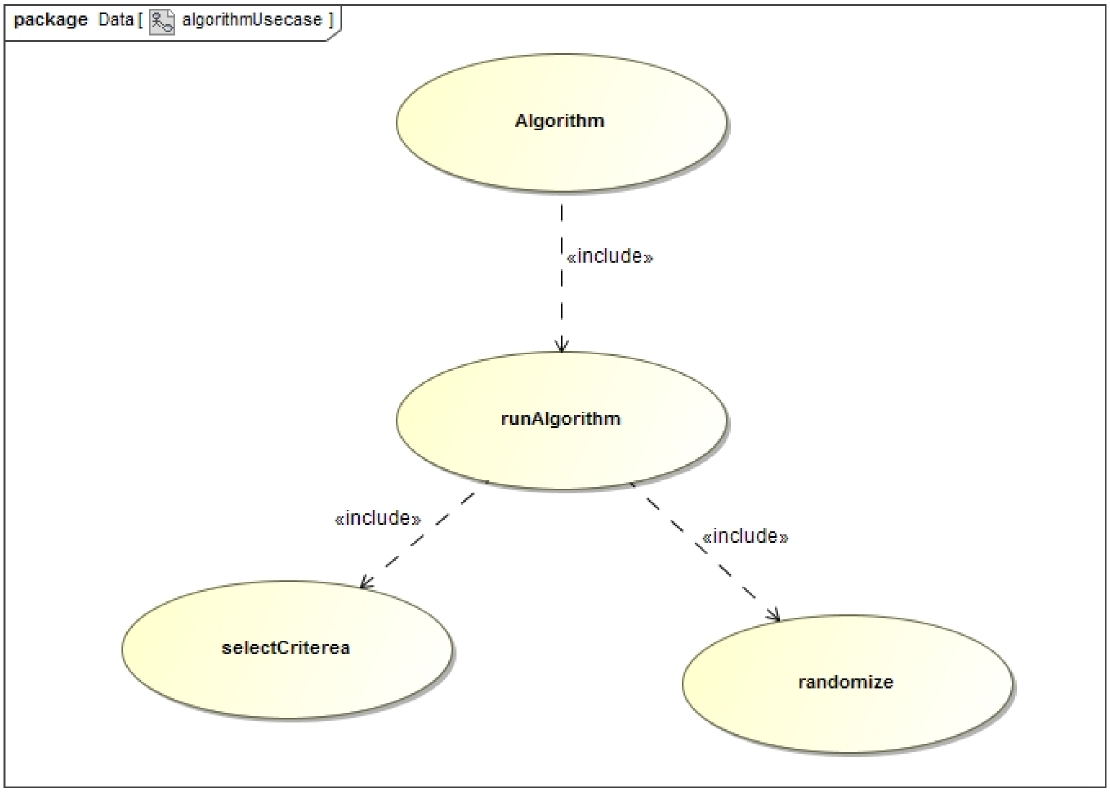

Figure 8: Algorithm module use case

2.5.1 Use-cases

The algorithm module adds the functionality required to shuffle subjects into teams.

1. RandomShuffle

   Priority: Critical.

   Pre-condition: Project must have users.

   Pre-condition: Team size or number of teams must be specified.

   Post-condition: Random teams are built.

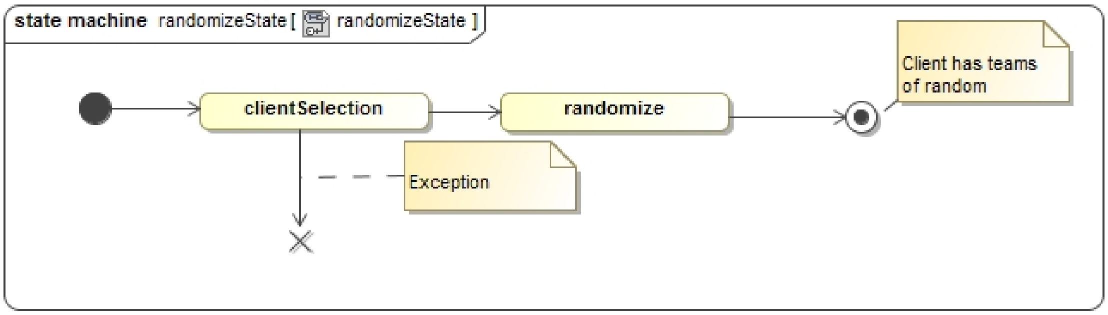

Figure 9: Randomize state diagram

2. SimilarShuffle

   Priority: Critical.

   Pre-condition: Project must have users.

   Pre-condition: Team size or number of teams must be specified.

   Pre-condition: Criteria with weights should be selected.

   Pre-condition: Similar shuffle should be specified.

   Post-condition: Similar teams are built.

3. DiverseShuffle

   Priority: Critical.

   Pre-condition: Project must have users.

   Pre-condition: Team size or number of teams must be specified.

   Pre-condition: Criteria with weights should be selected.

   Pre-condition: Diverse shuffle should be specified.

   Post-condition: Diverse teams are built.

2.6 Reporting Module

To be completed in future iterations

3 Architectural Requirements

This section discusses the software architecture requirements that is the requirements around the software infrastructure within which the application functionality is to be developed. The purpose of this infrastructure is to address the non-functional requirements. In particular, the architecture requirements specify

- the architectural responsibilities which need to be addressed,
- the access and integration requirements for the system,
- the quality requirements, and
- the architecture constraints specified by the client

3.1 Access and Integration Requirements

This section discusses

1. the requirements for the different channels through which the system can be accessed by people and systems, and
2. the integration channels which must be supported by this system. This section specifies the different channels through which users will be able to access the system services.

3.1.1 Human Access Channels

The system will be accessible by human users through the following channels:

- From a web browser through a rich web interface. The system must be accessible from any of the standards-compliant web browsers including all recent versions of Mozilla Firefox, Google Chrome, Apple Safari and Microsoft Internet Explorer.
- From mobile devices if time allows it.

3.1.2 System Access Channels

- Other systems should be able to access services offered by STORM. Perhaps direct feed of teams into other systems.
- Various technologies will be used such as:
  - Node.js
  - Express.js
  - EJS
  - Mongoose
  - SMTP

3.1.3 Integration channels

The system will allow manual integration through importing and exporting of CSV files. In particular, the system will support

- Importing of subject data from CSV files.
- Importing of new criteria for shuffling subjects from CSV files.
- Exporting of groups to CSV files.
- Exporting of subject data for offline editing to CSV files.

3.2 Quality Requirements

The quality requirements are the requirements around the quality attributes of the system and the services it provides. Quality requirements relevant to project STORM are listed below in order of priority.

3.2.1 Usability

Usability is one of the most important quality attributes. Usability tests should be performed to check whether:

- users find any aspects of the system cumbersome, non-intuitive or frustrating.
- the user has a positive experience and finds the functionality easy to use and learn.

3.2.2 Scalability

The system must implement a very generic optimization algorithm in order to be used by different parties in different contexts. In addition to different environments the system should be able to optimize groups for an extremely large number of subjects.

3.2.3 Performance

- The optimization algorithm should be able to sort a maximum of 10 000 subjects into groups and respond within 10 seconds.
- Reporting queries should respond within 15 seconds.

  **The above figures does not include the network round-trip which is outside the control of the system.*

3.2.4 Reliability

The system should provide by default a reasonable level of reliability and should be deployable within configurations which provide a high level of availability, supporting

- fail-over safety of all components and
- a deployment without single points of failure.

Hot deployment of new or changing functionality is not required for this system.

3.2.5 Deployability

The system must be deployable

- on Linux servers,
- and in environments using different databases for persistence of the STORM data.

3.2.6 Security

Security is a fairly important aspect of the system in order to protect sensitive project and subject data. The system should also allow users that are logged on to the system, to assign collaborators to a project.

3.3 Architectural Responsibilities

The architectural responsibilities for STORM includes the responsibilities of providing an infrastructure for

- a web access channel,
- hosting and providing the execution environment for the services/business logic of the system,
- persisting and providing access to domain objects,
- logging,
- and delivering reports readable by 3rd party applications,
- sending emails.

3.4 Architecture Constraints

The choice of architecture is largely unconstrained and the development team has the freedom to choose the architecture and technologies best suited to fulfill the non-functional requirements for the system subject to:

1. The architecture being deployable on a server, such as the Linux server of the University of Pretoria.
2. The architecture is constrained to using web technology and open source API's and tools.

4 Architecture design

4.1 Overview

MVC architecture - using express.js framework

1. Model - Database connection and access done via mongoose, requests handeled by AJAX.
2. View - ExpressJS defines a response object for rendering html views and serving the view with JSON data.

3. Controller - Defined in form of routes. ExpressJS routes are middleware functions which accepts request and result objects used to route data.

4.2 Modularization

The software architecture of STORM is a modular software architecture with a number of core modules and a number of pluggable add-on modules. Further add-on modules can be added at a later stage. Add-on modules may add additional functionality and may enrich excisting functionality through interception.

4.3 Architectural tactics addressing quality requirements

This section discusses the architectural tactics which are used to concretely address the quality requirements for the application

4.3.1 Database abstraction

The system includes a database abstraction module to improve:

- Deplayability as mentioned in section 3.2.5 in environments where different databases are used,

- Security as mentioned in section 3.2.6 to isolate database and hide sesitive subject data.

4.3.2 UI components framework

The system will use a rich, dynamic JQuery-UI component library in order to:

- provide a rich, dynamic user interface for usability (requirement 3.2.1),

- improve scalability (requirement 3.2.2) and performance (requirement 3.2.3).

4.3.3 Maximize client-side scripting

The system will use client-side javascript where possible to address:

- Performance (requirement 3.2.3) by avoiding the netork round trip,
- scalability (requirement 3.2.2) and security (requirement 3.2.6) by hiding data out of a user's bounds.

4.3.4 MVC

The system will use a MVC pattern to address:

- Usability (requirement 3.2.1),
- Deployability (requirement 3.2.5).

4.4 Architectural Components

This section discusses the architectural components, technologies and frameworks used to address the architectural responsibilities and the architectural tactics chosen to address the quality require- ments as specifed in section 3.2.

4.4.1 JavaScript

JavaScript is chosen as a single programming language used across the presentation and business logic layers of the system in order to implement the tactic of minimizing the technology suite. This prototype based dynamic programming language which supports duck typing is aligned with realizing flexibility and rapid development. Using a single programming language across the client and the server reduces complexity and improves maintainability.

4.4.2 Node.js

The system will be deployed in a Node.js execution environment. Node.js implements the asynchronous-processing by providing a framework for event-driven asynchronous callbacks in the form of asynchronous libraries. These typically provide second order functions which receive two functions as arguments.

```
1 provider.someFunction(task, callbackFunction);
```

The first is the function which may require waiting for resources or consume a significant amount of time and the second is the callback function which is to be called once the results from the first function are obtained.

4.4.3 MongoDB

The software architecture will use as persistence provider the MongoDB cross-platform document store. The reasons for this are that the application largely stores data whose state does not change (subjectID's, subjectNames). Using MongoDB as a NOSQL document store results in a highly cachable persistence environment which is very scalable (requirement 3.2.2) and can result in high levels of performance (requirement 3.2.3). The default locking mechanism is a readers-writer lock which allows concurrent read access but only allows a single write operation, i.e. while one transaction is writing a document no other transaction can read that document or write to that document. Scalability and performance can be further improved by using:

- Effective indexing
- Clustering with load balancing
- Sharding.

4.4.4 NodeMailer

STORM will make use of the NodeMailer JavaScript email client to send emails to a mail server.

4.4.5 jsreport Reporting Framework

The jsreport reporting framework will be used to provide a simple, yet powerful and flexible reporting framework for the application.