Camp Application and Management System (CAMs) Project Report

Declaration of Original Work for CE/CZ2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	Course	Lab Group	Signature/Date
REDACTED	REDACTED	REDACTED	REDACTED
REDACTED	REDACTED	REDACTED	REDACTED
REDACTED	REDACTED	REDACTED	REDACTED

Abstract
This report provides an overview of the design considerations and Object-Oriented Programming (OOP) principles used in the development of the Camp Application and Management System (CAMs) Command Line Interface (CLI) application. We also go over some of the challenges faced while structuring the application, in particular for the entity to entity relationships and managing complex application-layer logic.

---

1. Introduction

Our implementation of CAMs was designed to follow OOP design principles for the overall structure and relations, such as **SOLID**, as well as following industry standard **design patterns and architectures** such as Model-View-Controller (MVC) for the entire application, as well Singleton and Repository pattern for specific sections of the application.

To that end, our implementation splits the application into separate packages: Models, Views, Controllers, Service, Stores, Interfaces, Enums, Utils. Each package is distinct in its functionality to ensure **separation of concerns**, which will be detailed in Section 3.1.1.

We also overcame design challenges pertaining to the presence of N-M cyclic relations between entity classes by using a repository pattern and reference management, detailed in Section 3.2.1.

We made the following assumptions during the development of the program:

1. There will exist no more than 2^{32} instances of any class that is associated with an Integer key
2. The CSV file will not be malformed or contain contradictory data
3. There are no multiplicities of the same username
4. The application is not to be thread-safe; there are no mutex locks on data mutators. It is therefore not possible to have multiple users logged in by design
5. For suggestions and enquiries, since there is an “edit” function, there is only one allowed per user per camp until it is deleted after resolution
6. The computer’s internal clock is consistent with the timezone of the camp data
7. The console supports some ANSI color codes

With the exception of (4), these limitations are easily remedied due to the use of good OOP principles during development.

2. OOP Concepts

2.1 Abstraction

Abstraction was widely used in the program. For instance, the **controllers** will perform a sequence of necessary high-level logical operations to align with the application requirements. An example can be found in Figure 2.1a. On the other hand, the **services** (which are used by the **controllers**), will perform lower level operations to realize the desired functionality, such as manipulating the Models through their getters/setters. An example of service can be found in Figure 2.1b. Each function in a **service** layer does one distinct task. Similarly, each function in the **controller** performs one application-level action. The application-level actions are the application requirements for CAMs (i.e. the user menus).

```
private static void deregisterCamp() {
    Integer id = null;
    do {
        id = utils.InputParser.parseInInteger(sc,
                "Enter the camp ID to deregister from. Enter 'C' to cancel. ", 0, Integer.MAX_VALUE,
                INPUT_MAX_ATTEMPTS, "C");
    } while (id == null);

    if (!campStudentService.existCamp(id)) {
        System.out.println("Camp does not exist.");
        return;
    }

    if (campStudentService.isPreviouslyRegistered(id)) {
        System.out.println("You previously deregistered from this camp. Not allowed to re-register.");
        return;
    }

    if (campStudentService.isAlreadyRegistered(id)) {
        if (campStudentService.isAlreadyCCM(id)) {
            System.out.println("CCMs may not deregister.");
            return;
        }
        if (campStudentService.deregisterFromCamp(id))
            System.out.println("Deregistration successful. You will not be eligible for re-registration.");
    } else
        System.out.println("You are not registered to this camp or does not exist.");
}
```

Figure 2.1a: Example of Controller operation for deregistering from camp

```
public boolean deregisterFromCamp(Integer id)
{
    // UNDO add camp id to student entity AND add student username to camp entity
    HashMap<Integer, Camp> camps = DataStore.getCamps();
    Camp camp = camps.get(id);
    if(existCamp(id))
    {
        Student s = (Student) AuthStore.getCurUser();
        camp.getRegisteredStudents().remove(s.getUserID());
        if((s.getCamps().contains(id)))
        {
            return s.removeCamps(id);
        }
    }
    return false;
}
```

Figure 2.1b: Example of Service operation for performing a specific task

2.2 Encapsulation

Encapsulation was used whenever possible, by setting the attributes of all objects with states or data to private (see Figure 2.2a), and providing getters/setters where appropriate. Furthermore, for particularly crucial stateless operations, such as ensuring that two sets are pairwise disjoint (i.e. current students and previous students), the setters will enforce this relation instead of exposing the set to direct manipulation.

```
public class Student extends User{

    /* Integer array of camp ids that this user is registered to */
    private ArrayList<Integer> camps;

    /* Integer array of camp ids that this user was registered to. camps AND prevCamps must be pairwise disjoint or
     * undefined behavior */
    private ArrayList<Integer> prevCamps;

    /* Integer of campid that this student is CCM of iff role == CCM. invalid otherwise. */
    private Integer committee;

    /* Points of student, valid iff role == CCM. invalid otherwise. */
    private int points;
```

Figure 2.2a: Example of Student class with attributes set to private

```
    public Integer getCommittee() {
        return committee;
    }

    public void setCommittee(Integer committee) {
        this.committee = committee;
    }

    public int getPoints() {
        if(this.getRole() != UserRole.CCM) return 0;
        return points;
    }

    public boolean setPoints(int points) {
        if(this.getRole() != UserRole.CCM) return false;
        this.points = points;
        return true;
    }
```

Figure 2.2b: Example of getters and setters used in Student class

2.3 Inheritance

Inheritance was used to create hierarchical relations between classes. For instance, how Student, CCM, and Staff inherit from User, or how SuggestionRequest and EnquiryRequest inherit from Request. By using inheritance, we utilize common properties and methods between the related objects and promote the use of polymorphism by sharing a common parent class.

```
public class SuggestionRequest extends Request {
    public SuggestionRequest(String id, Integer campID, String content) {
        super(id, campID, content);
        // TODO Auto-generated constructor stub
    }
```

Figure 2.3a: SuggestionRequest inheriting from Request

2.4 Polymorphism

We used polymorphism together with other good OOP principles, such as the controller classes calling on interfaces (elaborated more in Section 3.1.5). Overriding is also used to implement abstract methods declared in their superclass. For example, AuthStudentService (see Figure 2.4b) inherits from AuthService class and overrides the abstract login() and logout() method in Figure 2.4a.

```
public abstract class AuthService {

    public abstract boolean login(String uid, String password);

    public abstract boolean logout();
```

Figure 2.4a: Abstract methods login() and logout() in the abstract AuthService class

```
public class AuthStudentService extends AuthService {

    @Override
    public boolean login(String uid, String password) {
        // TODO Auto-generated method stub
        HashMap<String, Student> students = DataStore.getStudents();
        if(students.containsKey(uid))
        {
            Student s = students.get(uid);
            if (s != null) return isAuth(s, password);
        }

        return false;
    }

    @Override
    public boolean logout() {
        // TODO Auto-generated method stub
        AuthStore.setCurUser(null);
        return true;
    }
```

Figure 2.4b: AuthStudentService performing method overriding of login() and logout()

3. Design Principles and Patterns

We utilize various design principles in this project, mainly the SOLID design principles that are covered from Section 3.1.1 - 3.1.5.
We then cover some additional design considerations specific to this project.

3.1 SOLID Design Principles
3.1.1 Single Responsibility Principle (SRP)

The **singular responsibilities** of the important packages in CAMs are detailed below:

Model
The model classes represent the entities and their relationships/associations. For instance, “User” class, from which “Student”, “CCM” and “Staff” inherit. It encapsulates data-related operations, and ensures that the application's data and attributes are consistent and valid.

View
The view classes are responsible for displaying the user interface, in this case, on the command line. Importantly, the data passed to the view classes is considered immutable and there is no logic besides printing the data.

Controller
The controller classes implement all of the high-level logic corresponding to the application layer function (i.e. those on the user menu), as well as all the logic checks for said function. There are separate controllers for each User type. For instance, the StudentController has the method registerCamp, which contains only logic checks (i.e. isAlreadyRegistered, isPreviouslyRegistered) and finally the actual call to complete the task registerForCamp. These methods and logic checks are all implemented by their corresponding Service layer, which is responsible for the business logic and dealing with the Model classes directly. None of the low-level logic required to interface with Models is implemented in Controllers.
In doing so, we distinctly separate the responsibility of the controller from data manipulation of the models such that, for instance, if the committee members can deregister, it will be a single line to remove, rather than rewriting the logic.

Controllers can be thought of as similar in function to middleware or library.

Services
The service classes implement the business logic of lower-level functions, such as interacting with setters and getters in the entity (Model) classes, so as to provide a single logical function for the associated Controller class.

Services can be thought of as similar in function to drivers (i.e. R/W ops -> R/W to entity).

3.1.2 Open-Closed Principle (OCP)

Our implementation allows for **easy extension of the existing code,** through the use of interfaces and abstract classes. For instance, the interface IRequestService can be used by new implementations without changes to the existing code. The abstract AuthService class is also extended by AuthStudentService, AuthStaffService, as they implement the abstract login() and logout() methods.

3.1.3 Liskov Substitution Principle (LSP)

We utilize LSP by ensuring that derived classes are **substitutable for their base classes**. This is done in part by ensuring whenever a part of the program uses subtyping, that its subtypes have pre-conditions no stronger than the base class, and post-conditions no weaker than the base class. In particular, given that the program logic expects some class Z from which class A, B inherit, then it will work on all Z and its subtypes A,B. For instance, the use of upper-bounded ArrayList<? extends Request> to pass all subtypes of Request to be displayed on CLI since they do not “remove” features over the base class.

3.1.4 Interface Segregation Principle (ISP)

We follow the ISP by **ensuring that interfaces are not bloated**. For instance, there are interfaces IStudentCampService and IStaffCampService which require implementations of specific actions, rather than a more generic IUserCampService. IRequestService adheres to this as it is specific in its purpose by focusing on aspects of request management, such as creating, deleting and editing requests.

3.1.5 Dependency Inversion Principle (DIP)

In order to promote loose coupling of the code, we ensured that **dependencies target interfaces and abstract classes** whenever possible, rather than their implementers.

For instance, StudentController depends on IStudentCampService rather than its concrete implementation StudentCampService. The same is the case for uses of IRequestService and its implementers EnquiryRequestService and SuggestionRequestService, or IAuthService for its user-specific implementers.

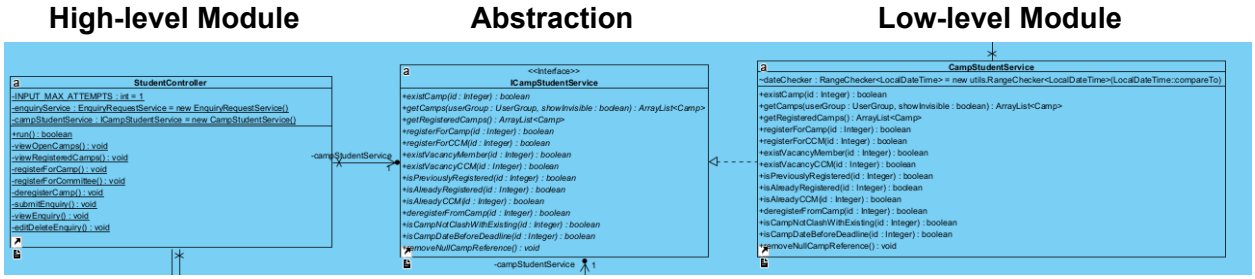


Figure 3.1.5: Illustration of DIP in CAMs

3.2 Other design considerations

3.2.1 Repository Design Pattern and Object Lifecycle Management

During the design of CAMs application, we observed that there exists significant interconnecting relations between entity classes. In particular, we find that there are two **issues**:

- N-M cyclic relations between objects (i.e. User can have Camp, Camp can have User)
- Significant requirement for the application to iterate through all associations for some object (i.e. all Camps of some User or all Users of some Camp)

Suppose that we keep only one direction of the required relations (i.e. 1-M). Then any relation in the other direction (i.e. for any of M objects), may be found in O(N*d) time, where d is the time complexity of the underlying linear search. The cyclic nature of the relations also adds difficulty. Hence, due to the prevalence of such operations, we find that it is necessary to fully represent the bidirectional relations, in order to improve the time complexity of accesses, as well as to decrease the potential for logical/consistency errors.

Therefore, we **formalize** the **requirements** of the program logic for entity relations:

- We preserve N-M relations to ensure O(n) time complexity in both directions, for any listing of all n associations. For instance, this means that any entity Camp holds reference to the set of associated Users, and vice versa.

This introduces a data integrity issue. Suppose that two instances of Users hold reference to the same Camp entity. Then it is not clear how Camp may be safely deleted; if some reference is not set to null, then “ownership” will effectively be passed to that reference and the GC will never destroy the instance (since the lifetime of the User classes is equal to that of the program). The requirement for doing so is to either keep track of all existent references, or iterate through all possible entity classes that may contain such a reference and set each to null. Furthermore, such a design would cause it to be ambiguous as to where some particular instance should be accessed from.

As such, we **add** on the **requirements**:

- Every instance of an entity class has its reference held only at a central DataStore class, such that all accesses to the entity can happen only through that class, and that nullifying reference to the instance is guaranteed to allow GC reachability to determine that it may be destroyed.

Therefore, we use a Repository design pattern, where all instances of entity classes are stored in the repository class “DataStore”. To resolve the issue of GC reachability (particularly for deletion), we borrow concepts of “ownership” from languages like C++ and Rust, by allowing

only the DataStore (the “owner”) to hold references to the entities, and the relations between entity classes to be modeled by unique ID “references” stored within the entity classes. For instance, Camp will not store ArrayList<User> but ArrayList<Integer>, where each (Integer,User) K,V pair is stored in HashMap<K,V> within the DataStore. We considered some alternatives, such as use of weak or soft references, but our proposed method was ultimately implemented due to ease-of-use in the program logic.

We handle the issuance of Integer IDs by sequentially issuing them. We stated previously in Section 1 that this implementation is limited to 2^{32} of any class instances, which is sufficient for the scope of this application.

4. Test Cases and Results

User Functionality

Test case	Test Description	Expected Result	Pass/ Fail
1.	Error case: Cannot login: invalid user or valid ID but wrong password	“Invalid credentials. Try again”	Pass
2.	Successful login	Prompt user to change password for first login. Display menu list depending on role of user (Staff/ student/ CCM)	Pass
3.	User changes the default password. Relog in with new password	Password changed, user has to log in with new password.	Pass

Student Functionality

Test case	Test Description	Expected Result	Pass/ Fail
1.	Login after changing password	Successful login, display Student Menu	Pass
2.	View open camps	All visible camp information displayed	Pass
3.	View registered camps	Camps registered by user will be displayed	Pass
4.	Register for camp (as participant)	Registration successful	Pass
5.	Deregister from: Error case: a.) Camp that user is NOT a	a.) Deregistration unsuccessful, user not registered to the camp	Pass

	participant of Normal case: b.) Camp that user is a participant of	b.) Deregistration successful, student not eligible for re-registration	
6.	Register for camp that previously deregistered from	Not allowed to re-register	Pass
7.	Submit enquiry	Enquiry submitted	Pass
8.	View enquiry	Able to view response status, response, as well as content	Pass
9.	Edit/delete enquiry.	Enter '0' to edit, '1' to delete 0.) Enquiry edited 1.) Enquiry deleted, unable to see the enquiry even if View Enquiry is selected	Pass
10.	Error case: Edit/delete enquiry. "A" inputted	Invalid input. Please enter an integer.	Pass

CCM Functionality

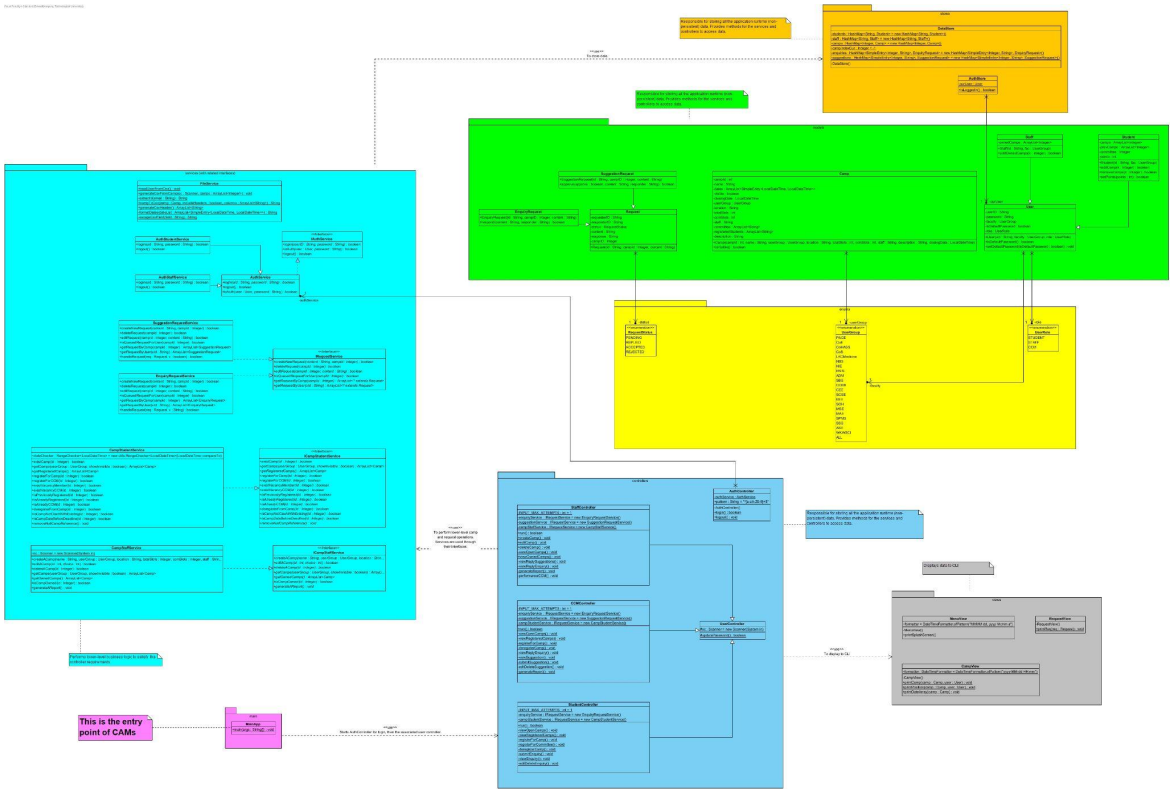
Test case	Test Description	Expected Result	Pass/ Fail
1.	Login after changing password	Successful login, display CCM Menu	Pass
2.	From student menu: Register for camp (as committee)	Registration successful	Pass
3.	a.) Deregister from camp that user is CCM of b.) Deregister from camp that user is participant of	a.) Unsuccessful. CCMs may not deregister b.) Deregistration successful, student not eligible for re-registration	Pass
4.	View/Reply student enquiries	Enter '0' to view enquiries, enter '1' to respond to enquiry	Pass
5.	New suggestion	Submit new suggestion, CCM gains 1 point	Pass
6.	View Suggestion a.) Error Case: CCM did not submit suggestion b.) Normal Case: CCM submitted suggestion	a.) Unsuccessful. CCM did not submit any suggestion b.) CCM can view his submitted suggestion	Pass
7.	Edit/Delete suggestion	Enter '0' to edit, '1' to delete 0.) Suggestion edited	Pass

		1.) Suggestion deleted, unable to see the suggestion even if View Suggestion is selected	
8.	Generate report	Report is generated	Pass

Staff Functionality

Test case	Test Description	Expected Result	Pass/ Fail
1.	Login after changing password	Successful login, display Staff Menu	Pass
2.	Create Camp. <u>a.) Error case:</u> Enter the Camp name :SCSE TOP Enter user group: SCSE Enter location: Hive Set max committee: abc <u>b.) Correct case:</u> Enter the Camp name :SCSE TOP Enter user group: SCSE Enter location: Hive Set max committee: 12 Set max slots: 6 Enter the camp description: Transition Orientation Program for SCSE Please enter a valid closing date and time in the format yyyy-MM-dd HH:mm: 2024-01-01 23:59 Please enter a valid date and time in the format yyyy-MM-dd HH:mm Starting date: 2024-01-10 09:00 Ending date: 2024-01-12 17:00	a.) “Invalid input. Please enter an integer.” b.) “Camp successfully created!”	Pass
3.	Edit Camp 2. Camp location Enter new camp location: ARC	Camp successfully updated, camp location changed to ARC	Pass
4.	View Open Camps	All camp information displayed	Pass
5.	View Owned Camps	Owned camp information displayed	Pass
6.	View/Reply Suggestions	Enter ‘0’ to view suggestions Enter ‘1’ to respond to a suggestion	Pass
7.	View/Reply Enquiry	Enter '0' to view enquiries. Enter '1' to respond to an enquiry. .	Pass
8.	Generate report	Report is generated	Pass

5. UML Diagram (Also available as image or VP diagram)



6. Reflection

We encountered difficulties during this project, primarily when ensuring separation of concerns. We used a MVC design pattern for the overall application to overcome this, and segment the various functionalities into separate packages such that there are minimal interdependencies.

Additionally, the relations between entity classes as described in Section 3.2.1 also brought about difficulties, which we resolved by using a Repository pattern.

Based on the above mentioned difficulties, we learned how to resolve them. We also learned various programming conventions such as Javadocs, as well as collaborative software version control using GitHub. Lastly, we learned how to apply the SOLID design principles in a project based setting.

Some further improvements include making the CLI more aesthetic (or intuitive), in particular by using a console that supports ANSI escape sequences for clearing the screen or placing a blinking cursor, or otherwise implementing a basic GUI inside the (currently) rather empty View classes. We could also implement additional features like multiple Request queueing, or compound keys for HashMap to further improve search times over both Camp and Student etc., possible with relatively little effort due to the OOP principles that were followed during development.