Programing2.5 - Course Project

Ali Shahrestani

⚠️ **IN THE EVENT OF PDF ISSUES, THE REPORT CAN BE ACCESSED AT** https://www.notion.so/Programing2-5-Course-Project-1dacdb148ce5804ab568e306c8144240 **IN ITS INTENTED/ORIGINAL VIEWING FORM.**

Table Of Contents

Instructions

▼ Deliverable 1 Instructions

Submit a project idea, addressing the following points:

- Scenario: Explain the scenario under which your project will operate.
- Design Paradigm: List the functionalities you plan to demonstrate.
- Expected Output: Describe the expected results and the actions the user can perform with your application.
- Specify the hierarchies (at least two) of the project
- Specify the what the interface is and why do you need it in your project
- Specify which methods apply runtime-polymorphism
- Specify in which class and for what purpose textIO will be used.
- Specify in which class(es) `Comparable` will be implemented, and what class(es) need `Comparator`
- A class diagram to show the classes and interfaces of the project and their relationship.
- Specify which part (class, interface and methods) will be implemented for deliverable 2 (50%)
- Git Repository: Initialize a Maven project with valid `.gitignore`, and a `README.md` file for a project description. Create a `doc` folder which contains Deliverable 1 PDF.

▼ Deliverable 4 Instructions

- Cover page with your name, project title, and course name.
- Outline/Table of Contents.
- Project Description (same as Deliverable 1).
- Program Features and Screenshots (showing how the project meets the requirements, with output and execution examples).
- Challenges (any unimplemented features or issues faced during development).

- Learning Outcomes (what you gained from the project).
- The report MUST be uploaded to both Lea and Git repository (in the `doc` folder)

Deliverable 1 - Project Description

Scenario

The project is called “Organization Budget Tracker”.

It is a fun poke at the financial operations that needed to be done when organizing VanierHacks 2025. This project is a much simplified version of different software used to track expenses and reimbursement.

The scenario/purpose of this project is for any organization (eg. a club or an association) that needs to buy things and reimburse those who actually made the purchases, all whilst keeping track of how much of the budget remains.

The program will be used by everyone in the organization but certain actions are reserved only for certain positions. For example, anyone can submit a request for reimbursement but only the treasurer can approve the transfer of funds.

The user must login to use the program.

Design Paradigm

The current planned functionalities (subject to change) are:

System Administrator

- Manage user accounts
  - View all users
  - Create a new user
  - Delete a user
  - Change a user’s type
- Configure sources

    - Create accounts
    - Create projects
    - Create budget
- View technical details (extra)
- View system audit logs (extra)

Treasurer

- Manage sources
    - View all accounts and their details
    - View all projects (events that need funding)
    - View all budgets (income sources) and their details
    - Update the amount of a budget (extra)
    - Edit projects details (extra)
- View financial audit logs (extra)
- Manage reimbursement
    - View all reimbursement requests
    - Approve reimbursement requests
    - Edit reimbursement requests (extra)

Organizer

- View their account
    - View the history of all approved reimbursements deposited into their account
- Request reimbursement
    - View the status of the pending reimbursement
    - View history of reimbursement
    - Modify pending reimbursement requests (extra)
- View all active events

Some functionalities overlap who can do them.

The functionalities marked as “extra” are low priority ones and the decision to implement them or not will be made when the other functionalities have been made.

Expected Output

All users will interact with the program with through the console. There is no graphical user interface.

User input will be heavily facilitated by the separate Java Scanner Menu Framework (provides input/error handling and prompt reuse).

The program starts with a default sys admin account. The sys admin must then create accounts for all other users.

Users must first login before being able to perform actions outlined in the previous section.

Users will be able to submit reimbursement request and track their status.

The treasurer will be able to view pending requests, approve/reject them, and monitor the budget.

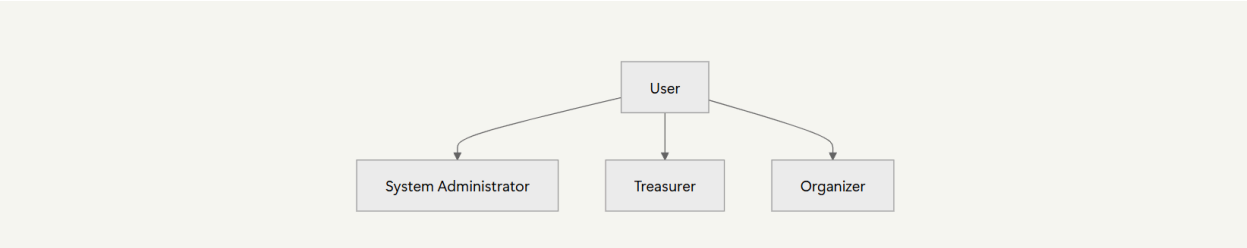
The program’s data will be saved in different folders and files. The data will be loaded when the program starts or when needed by their related actions.

The program determines which organization to load based on the first command line argument passed which specifies the location of the organization’s files.

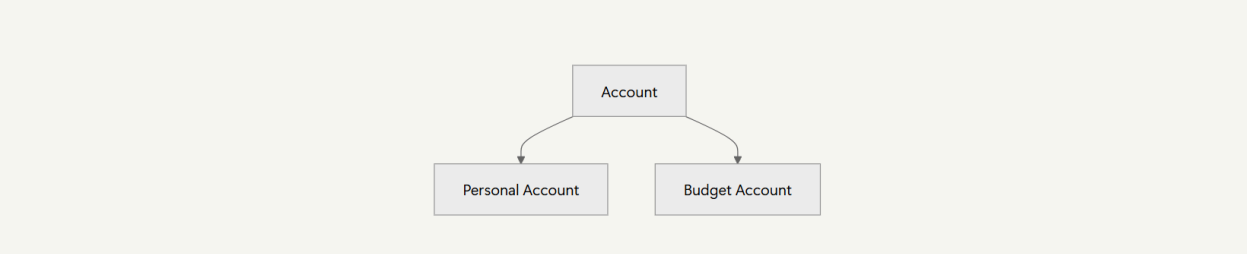
Concepts

Hierarchies

User Hierarchy



Account Hierarchy



There will be more classes (eg `Project`, `Transaction`, `Expense`, `Reimbursement`, etc).

Interfaces

Transactable

This will be an interface with `

The `Savable` interface will declare a `export()` and a `import()` method to be used by the `DataManager` or its subclasses. This will be implemented by almost all the object classes to self-implement the necessary steps to save/load the object.

There may be more interfaces implemented later on.

Polymorphism

Polymorphism will be demonstrated in the `displayUserInfo()` method. This method will be defined in the `User` class and overridden in the subclasses to display user-

specific information (eg regular organizers have their personal accounts, there will be info on that).

There will be more instances of polymorphism.

TextIO

TextIO will be used in the `DataManager` class (and the subclasses pertaining to different parts of the system).

The class will handle reading, loading, and writing user data, reimbursement records, and budget information for data persistence.

The class is also crucial for initial program startup to load information about the organization and its users.

Some compartments of the program will save data in a single file.

Some compartments of the program will save data per line in a file.

Some compartments of the program will use a whole directory and separate data into individual files.

Comparison

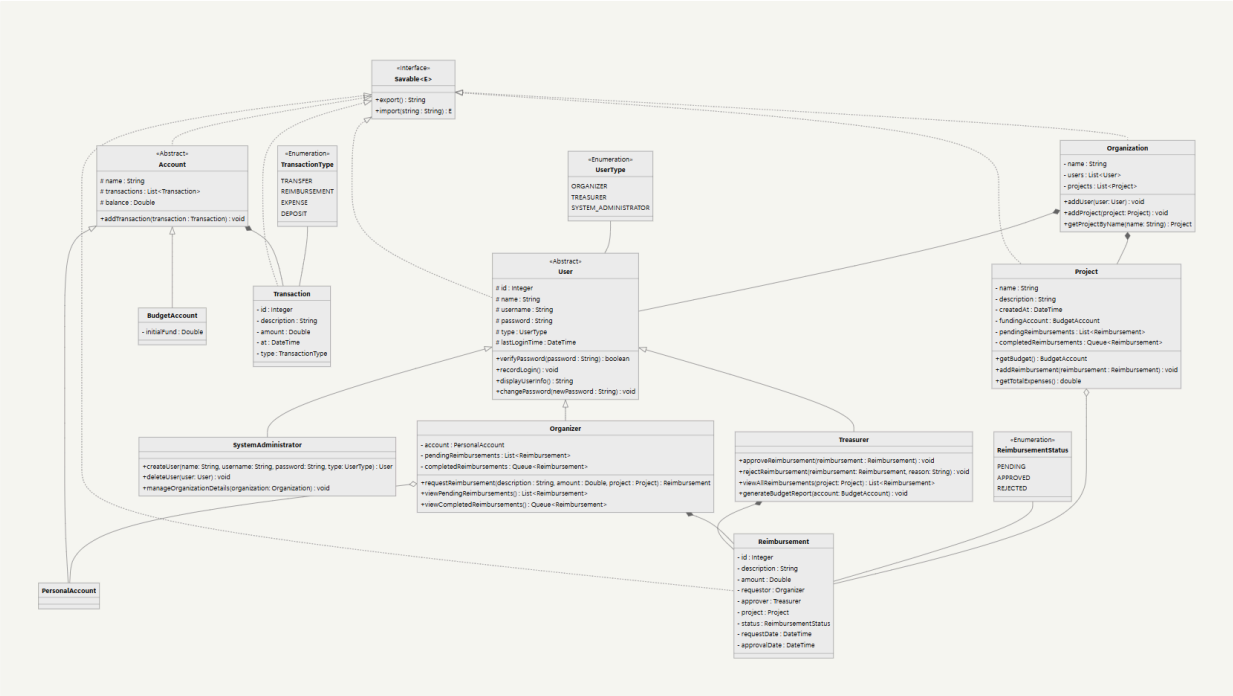
The `User` class will implement the `Comparable` interface to be able to sort users based on their permission level in the organization.

The `Project` class will implement the `Comparator` interface to be able to sort projects based on different criteria such as:

- Amount of money used on the project
- Number of reimbursement for the project
- Number of unique people reimbursed for the project

There may be more comparisons implemented.

Class Diagram



This diagram can be viewed as an image in mermaid-diagram-2025-04-19-212150.png

Note the following implications of the UML diagram above:

- Auto generated methods such as constructors, getter/setters, equals, etc are not included
- Abstract methods are only included in the abstract class
- Abstract interface methods are only included in the interface
- Quantifiers (cardinality / multiplicity) are not defined
- The diagram makes no guarantees on the order of fields and methods
- Utility classes are not showed
- User intractability (”flowchart”) classes are not showed
- Additional annotations follow https://mermaid.js.org/syntax/classDiagram.html

A full class diagram will be available for subsequent deliverables. The diagram above is subject to change and is only an initial plan.

Priority for the 50% milestone is to implement `User` and its subclasses, `Account` and its subclasses, `Transaction` and `Project`.

To further clarify, creating the conceptual classes and their methods (business logic) will be prioritized over “flowchart” classes and their methods (user intractability with the program). Data preservation will be the third area of concern.

And all Enums related the classes mentioned.

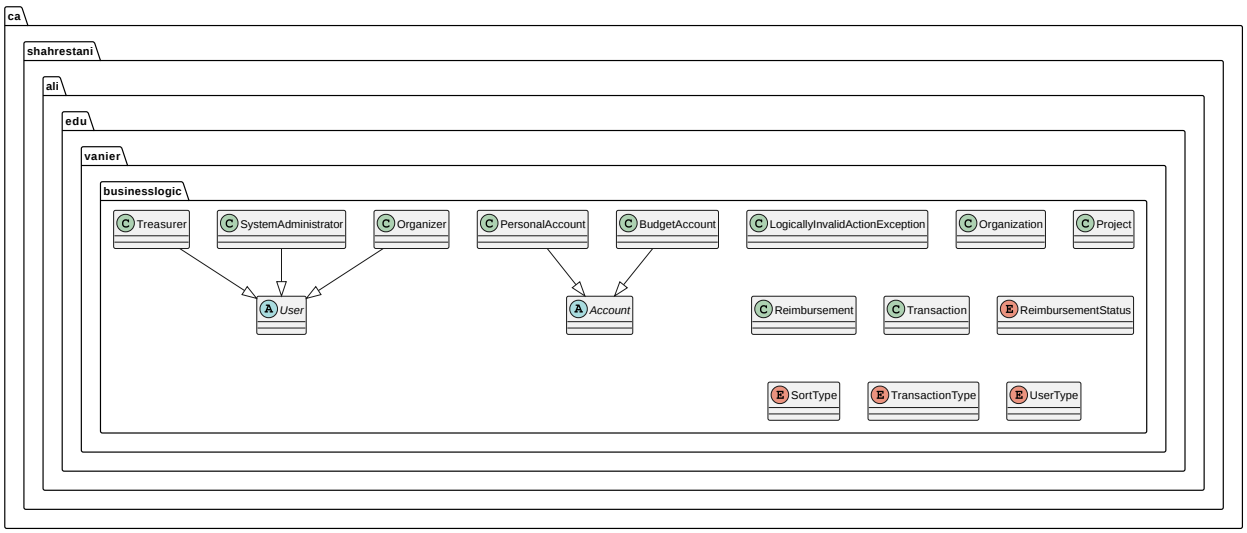
GitHub

The project will be stored on GitHub at https://github.com/VanierCollege/Programing2.5-CourseProject/.

Deliverable 4 - Project Report

Program Features

OOP



The project uses multiple types of objects to separate business logic into related groups.

Notably,

1. Interfaces such as `SavableFactory` and `Savable` .

```
package ca.shahrestani.ali.edu.vanie

import java.util.Map;

/**
 * Standardize method to allow class
 */
public interface SavableFactory<T ex
    /**
     * Method to construct a new instar
     *
     * @param str the data string
     * @return the newly constructed c
     */
    T load(String str, Map<String, Obje
}
```

```
package ca.shahrestani.ali.edu.vanie

/**
 * Standardize methods to allow class
 */
public interface Savable {
    /**
     * Method to allow classes to defin
     *
     * @return a string that represents
     */
    String export();
}
```

These interfaces allows business classes to only have to handle their own data parsing (containing responsibility) while allowing the `DataManager` to dynamically call these methods regardless of which class it is. The interfaces were also useful because of their ability to be referenced in generics as type bounds.

This also showcases the application of method polymorphism and overriding in the project as each class implemented its methods differently.

```
public static <T extends Savable> T loadSavable(String type, String data, Map<S
        SavableFactory<T> factory = (SavableFactory<T>) factories.get(type);
        ...
```

2. Enums such as `UserType`.

```
package ca.shahrestani.ali.edu.vanier.businesslogic;

public enum UserType {
    ORGANIZER(5),
    TREASURER(2),
    SYSTEM_ADMINISTRATOR(0);

    final int level;

    UserType(int level) {
        this.level = level;
    }
}
```

This Enum has a constructor to associate a numerical value to the constants. The numerical value allows for sorting (defined in `User` comparators) users based on their type.

3. Abstract classes such as `User`.

Having the parent class be abstract allows the users hierarchy to share methods, fields, and interface contracts will all of its subclasses without having to worry about what “new User()” represents. The `User` superclass has `SystemAdministrator`, `Organizer`, and `Treasurer` as subclasses.

4. Unit testing for business classes.

Element	Class, %	Method, %	Line, %	Branch, %
businesslogic	89% (26/29)	55% (89/160)	51% (202/389)	34% (31/90)
SortType	0% (0/1)	0% (0/2)	0% (0/3)	100% (0/0)
Treasurer	100% (2/2)	25% (2/8)	6% (2/31)	0% (0/2)
PersonalAccount	100% (2/2)	42% (3/7)	23% (3/13)	100% (0/0)
SystemAdministrator	100% (2/2)	42% (3/7)	30% (4/13)	100% (0/0)
BudgetAccount	100% (2/2)	50% (4/8)	31% (5/16)	100% (0/0)
Transaction	50% (2/4)	52% (9/17)	45% (17/37)	50% (5/10)
Reimbursement	100% (2/2)	52% (11/21)	50% (32/64)	45% (9/20)
User	100% (2/2)	53% (8/15)	53% (14/26)	0% (0/12)
Project	100% (2/2)	55% (10/18)	51% (24/47)	16% (2/12)
Organization	100% (2/2)	57% (11/19)	75% (24/32)	100% (0/0)
Account	100% (2/2)	63% (12/19)	70% (42/60)	39% (11/28)
Organizer	100% (2/2)	72% (8/11)	61% (19/31)	66% (4/6)
LogicallyInvalidActionException	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
ReimbursementStatus	100% (1/1)	100% (2/2)	100% (4/4)	100% (0/0)
TransactionType	100% (1/1)	100% (2/2)	100% (5/5)	100% (0/0)
UserType	100% (1/1)	100% (3/3)	100% (6/6)	100% (0/0)

Coverage as of May 10 2025

There are over 70 unit tests — at the time of writing — attempting various possible situations including extreme ones to ensure that methods operate correctly.

Eg.

```
@Test  Ali
public void testAddExpense_example() {
    double balance = 3.00;
    double input = 1.00;
    BudgetAccount account = new BudgetAccount( name: "Test", balance);
    Project project = new Project( name: "Test",  description: "Testing Project", account);
    project.addExpense(input);
    Transaction transaction = account.getTransactionList().getFirst();
    Assertions.assertTrue( condition: account.getBalance() == balance - input
            && transaction.getType().equals(TransactionType.EXPENSE)
            && transaction.getAmount() == input
            && transaction.getDescription().contains(project.getName()));
}
```

5. Exceptions such as `LogicallyInvalidActionException` .

This exception represents actions that are invalid within the business logic context.

Eg.

```
public void recordCompletedReimbursement(Reimbursement reimbursement) {
    ...
    if (reimbursement.getStatus().equals(ReimbursementStatus.PENDING)) {
        throw new LogicallyInvalidActionException("Cannot record a pending reimb
    }
    ...
}
```

Patterns

1. Overloading methods.

Eg.

```
    public static <T extends Savable> T loadSavable(Class<T> type, String data, M
        return loadSavable(type.getSimpleName(), data, dependencies);
    }

    public static <T extends Savable> T loadSavable(String type, String data, Map
        ...
    }
```

Method overloading was used to provide shortcuts for common uses of a method.

Constructors being overloaded (aka multiple constructors) is also a crucial part of the program. Each class also has a simple arguments constructor that takes in the minimum amount of information to create a new instance of the class and one all-args constructor to be used when loading from saved data.

2. Leveraging diverse data structures.

Eg.

```
Map<String, SavableFactory<? extends Savable>> in DataManager
ZonedDateTime in User
Set<Transaction> in Account
ReimbursementStatus  in Reimbursement
List<String> in TransactionFactory
etc
```

Initially queues and stacks were used but it was deemed that a set's no-duplicates nature is the most important behavior throughout the system to ensure data isn't duplicated. Majority of data is thus stored in hash sets or linked hash sets.

3. Handling exceptions.

Eg.

```
Account account;
try {
    account = loadSavable(Account.class, rawAccount, null);
} catch (SwitchFactorySignalException sfsE) {
    account = loadSavable(sfsE.cls, rawAccount, null);
}
```

This specific try-catch block demonstrates the program's ability to load data while still keeping responsibility where it is needed. Since the `Account` superclass doesn't know about subclass specific fields, it cannot create a new instance by itself. However, the parent's loading implementation can still be responsible for initially parsing the raw data to determine which subclass to use. This avoids `DataManager` having to know how-to and check business specific implementation details.

4. Utilizing streams.

Eg.

```
private void assertEqualsCommaList(List<String> expected, List<String> actual)
    expected = expected.stream().map(str -> str.replace(",", "$")).toList();
    actual = actual.stream().map(str -> str.replace(",", "$")).toList();
```

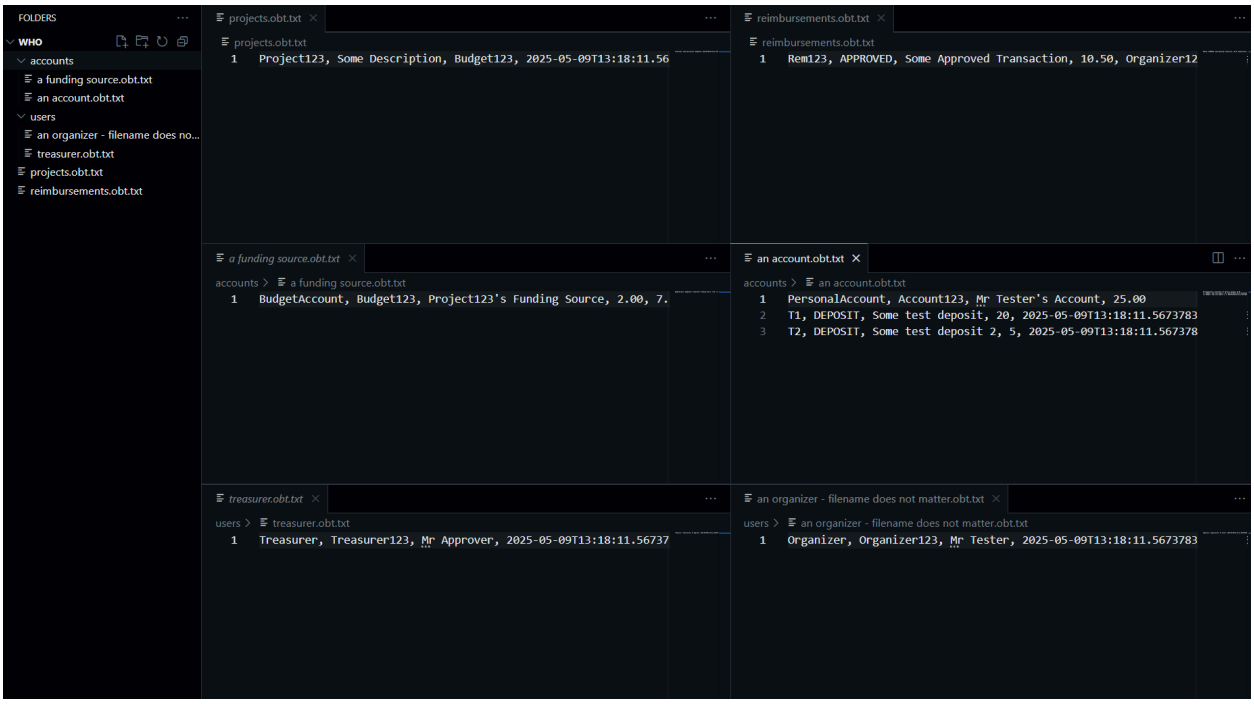
```
    Assertions.assertEquals(expected, actual);
}
```

A stream is used to help distinguish whether a comma being printed into the console is there because it is separating list elements or if its a literal comma inside a string. This was extremely useful in testing the `BracketAwareSplitter` util class because it splits comma separated values from raw strings.

TextIO

The `DataManager` util is the class responsible for reading and saving the program's data.

All data is stored as comma separated values. `Account` also goes a step further by having the first line be account data itself (header) and subsequence lines store the transactions of the account.



Given a valid structured folder and data files being present (shown above), the program can successfully read and load all the saved data (shown below).

File - Main

```
Organization{name='WHO', users=[Treasurer{id='Treasurer123', name='Mr Approver', type=TREASURER, createdAt=REMOVED FOR
SCREENSHOT, lastSystemAccess=REMOVED FOR SCREENSHOT}, Organizer{id='Organizer123', name='Mr Tester', type=ORGANIZER, createdAt
=REMOVED FOR SCREENSHOT, lastSystemAccess=REMOVED FOR SCREENSHOT}], accounts=[BudgetAccount{initialFund=7.0, id='Budget123',
name='Project123's Funding Source', transactionList=[], balance=2.0}, PersonalAccount{id='Account123', name='Mr Tester's
Account', transactionList=[Transaction{id='T1', type=DEPOSIT, description='Some test deposit', amount=20.0, transactedAt=
REMOVED FOR SCREENSHOT}, Transaction{id='T2', type=DEPOSIT, description='Some test deposit 2', amount=5.0, transactedAt=
REMOVED FOR SCREENSHOT}], balance=25.0}], projects=[Project{name='Project123', description='Some Description', fundingAccount=
BudgetAccount{initialFund=7.0, id='Budget123', name='Project123's Funding Source', transactionList=[], balance=2.0}, createdAt
=REMOVED FOR SCREENSHOT}], reimbursements=[Reimbursement{id='Rem123', status=APPROVED, description='Some Approved Transaction
', amount=10.5, requester=Organizer{id='Organizer123', name='Mr Tester', type=ORGANIZER, createdAt=REMOVED FOR SCREENSHOT,
lastSystemAccess=REMOVED FOR SCREENSHOT}, treasurer=Treasurer{id='Treasurer123', name='Mr Approver', type=TREASURER, createdAt
=REMOVED FOR SCREENSHOT, lastSystemAccess=REMOVED FOR SCREENSHOT}, project=Project{name='Project123', description='Some
Description', fundingAccount=BudgetAccount{initialFund=7.0, id='Budget123', name='Project123's Funding Source',
transactionList=[], balance=2.0}, createdAt=REMOVED FOR SCREENSHOT}, requestedAt=REMOVED FOR SCREENSHOT, approvedAt=REMOVED
FOR SCREENSHOT}]}
[BudgetAccount{initialFund=7.0, id='Budget123', name='Project123's Funding Source', transactionList=[], balance=2.0},
PersonalAccount{id='Account123', name='Mr Tester's Account', transactionList=[Transaction{id='T1', type=DEPOSIT, description='
Some test deposit', amount=20.0, transactedAt=REMOVED FOR SCREENSHOT}, Transaction{id='T2', type=DEPOSIT, description='Some
test deposit 2', amount=5.0, transactedAt=REMOVED FOR SCREENSHOT}], balance=25.0}]
[Treasurer{id='Treasurer123', name='Mr Approver', type=TREASURER, createdAt=REMOVED FOR SCREENSHOT, lastSystemAccess=REMOVED
FOR SCREENSHOT}, Organizer{id='Organizer123', name='Mr Tester', type=ORGANIZER, createdAt=REMOVED FOR SCREENSHOT,
lastSystemAccess=REMOVED FOR SCREENSHOT}]
[Project{name='Project123', description='Some Description', fundingAccount=BudgetAccount{initialFund=7.0, id='Budget123', name
='Project123's Funding Source', transactionList=[], balance=2.0}, createdAt=REMOVED FOR SCREENSHOT}]
[Reimbursement{id='Rem123', status=APPROVED, description='Some Approved Transaction', amount=10.5, requester=Organizer{id='
Organizer123', name='Mr Tester', type=ORGANIZER, createdAt=REMOVED FOR SCREENSHOT, lastSystemAccess=REMOVED FOR SCREENSHOT},
treasurer=Treasurer{id='Treasurer123', name='Mr Approver', type=TREASURER, createdAt=REMOVED FOR SCREENSHOT, lastSystemAccess=
REMOVED FOR SCREENSHOT}, project=Project{name='Project123', description='Some Description', fundingAccount=BudgetAccount{
initialFund=7.0, id='Budget123', name='Project123's Funding Source', transactionList=[], balance=2.0}, createdAt=REMOVED FOR
SCREENSHOT}, requestedAt=REMOVED FOR SCREENSHOT, approvedAt=REMOVED FOR SCREENSHOT}]
```

Each yellow highlight corresponds to one of these prints:

```
System.out.println(organization);
System.out.println(organization.getAccounts());
System.out.println(organization.getUsers());
System.out.println(organization.getProjects());
System.out.println(organization.getReimbursements());
```

The important thing to emphasize — which is also what makes this program's features the most unique — is that although multiple objects reference the same objects (and overlapping) each object only exists once. Objects are only created when files are being read from the data directory.

If a class such as `Organizer` needs a reference to another class such as `PersonalAccount`, all it needs to do is export the account's ID. When loading `Organizer` will attempt to get the actual object reference by looking up the ID in the global map. This is possible thanks to Java's pass-by-value (value being the object's reference actually).

If saved data reference an invalid depended object, the program can identify this issue and (at the moment) skip loading the problematic data without effecting all other saved data.

```
Failed to load data of type Organizer: Organizer, Organizer123, Mr Te...
(IllegalArgumentException) Organizer (uID:Organizer123) references an invalid account (aID:Account000))
Failed to load data of type Reimbursement: Rem123, APPROVED, Some Approve...
(IllegalArgumentException) Reimbursement (rID:Rem123) references an invalid requester (uID:Organizer123))
```

After editing **Organizer123**'s data file to reference **Account000** instead of **Account123**, loading will fail because it will not be able to find **Account000**. Subsequently, **Rem123** will also fail to load because it depends on **Organizer123** to exist.

> ⚠️ To ensure the program does not operate with malformed or incomplete data, the effect of some missing data propagates to all other objects that depend on it. Despite this potentially causing a big impact stemming only from one corrupt file, it is safer to operate with this system. When user interactivity (frontend) is implemented, a report of failed data can be presented.
> The only major issue with this system is that if incomplete data is loaded and then the program data is saved, all data that wasn't loaded will be lost since the files will be overwritten. Thus it will be crucial to resolve incomplete data if present before continuing to use the program.

Data for `User` and `Account` are stored in individual files in their respective subfolders instead of being all in one file for the following reasons:

a. They are the foundational building blocks that allow all other types of records to exist so it best to keep each separate to avoid potential data mingling issues.

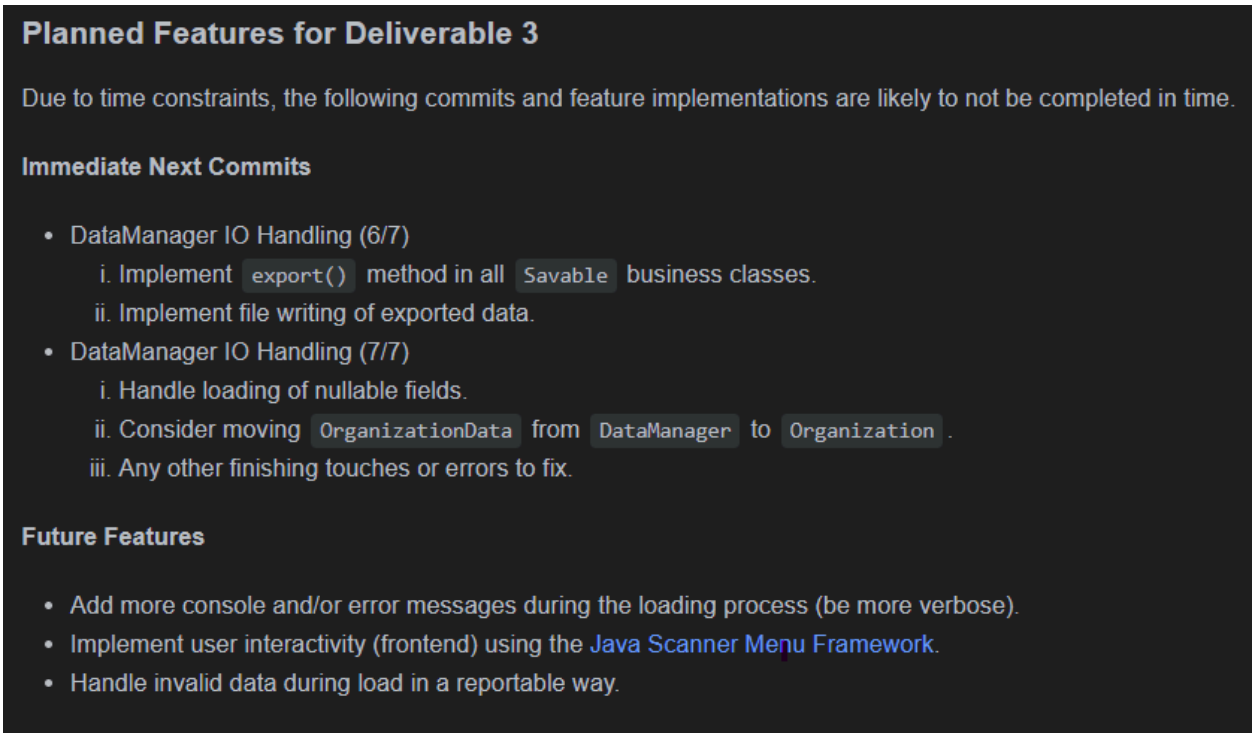
b. Makes it easier to share a user's data including their account without having to go looking in all the organization's data.

💡 A potential alternative would be to have `User` and `Account` be the ones stored in a single file and have the other types of data in separate files.

a. The program “needs” to load all users during startup to facilitate login anyways.
b. The program can only load data it needs without having to load all reimbursements, projects, etc. However, to make this possible some meta information will have to be stored in the filename so the filesystem can find the relevant files to lead without having to read them first.
c. This makes saving the other types of data (which are more common than the number of users/accounts) easier.
d. Avoid the incomplete data being overwritten issue. An organization’s reimbursement records are more important to be presented even if malformed until they are verified than replaceable users/accounts.

Challenges

The biggest challenge in this project is the features from the initial plan that weren’t completed.



Planned Features for Deliverable 3

Due to time constraints, the following commits and feature implementations are likely to not be completed in time.

Immediate Next Commits

- DataManager IO Handling (6/7)
  i. Implement `export()` method in all `Savable` business classes.
  ii. Implement file writing of exported data.
- DataManager IO Handling (7/7)
  i. Handle loading of nullable fields.
  ii. Consider moving `OrganizationData` from `DataManager` to `Organization`.
  iii. Any other finishing touches or errors to fix.

Future Features

- Add more console and/or error messages during the loading process (be more verbose).
- Implement user interactivity (frontend) using the Java Scanner Menu Framework.
- Handle invalid data during load in a reportable way.

The following challenges were minor:

- Ensure Intellij executes the program correctly by adding `--enable-native-access=ALL-UNNAMED` to the Java VM arguments.
- Ensure Intellij builds the program with all the dependencies.

Learning Outcomes

- Discover and learn how to use https://jitpack.io/ to include a GitHub repository directly as a project dependencies without having to go through the official process of publishing a package to central repositories.
- Help other students with questions related to their project and the best way to design certain aspects of a program.
- Articles such as:
  - https://reflectoring.io/howto-format-code-snippets-in-javadoc/
  - https://dev.to/cad97/what-is-a-lexer-anyway-4kdo
  - https://medium.com/@AlexanderObregon/javas-paths-get-method-explained-9586c13f2c5c

- https://plantuml-documentation.readthedocs.io/en/latest/formatting/all-skin-params.html
- https://www.uml-diagrams.org/class-reference.html
- https://openjdk.org/jeps/472
- https://openjdk.org/jeps/8305968
- https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/
- https://programmingduck.com/articles/defensive-programming

GitHub

The project can be accessed on GitHub at https://github.com/VanierCollege/Programing2.5-CourseProject/.

Appendices

Diagrams