Software Design - Assignment 2

Team Number: 25, Mentor: Linus Wagner

Name	Student Number	Email
Maria P. Jimenez M.	2692270	m.p.jimenezmoreno@student.vu.nl
Lennart K.M. Schulz	2734873	l.k.m.schulz@student.vu.nl
Dovydas Vadišius	2744980	d.vadisius@student.vu.nl
Laura I.M. Stampf	2701672	l.i.m.stampf@student.vu.nl

Contents

Note: in the following **Classes** will be written in bold, **objects** in underlined bold, class *attributes* and *methods()* in italics, static *members* and *methods()* in underlined italics, ***States*** in bold italics, and `message` in monospace.

1 Summary of Changes from Assignment 1

Author: Laura

After considering the feedback received from our mentor, we have updated three main sections from our initial project pitch: (1) the main types of users, (2) the functional requirements, and (3) the quality requirements of our system. Additional changes have been made due to decisions made during our diagram development process, however, these are not reflected in this section as they are not based on Linus' feedback. The updated version of our project pitch can be found on GitHub (here).

Software users are not a main user: Initially, we identified *software users* as one of the main users of our system. However, our discussion with Linus and his written feedback helped us realize that this is not the case. Although software users with little to no proficiency in coding may be interested in tinkering around with our system, they are by no means the **main** target audience and only represent a small minority of our system's users. Their interests are not essential to our system's development; thus, we have eliminated software users from our list of main users.

Requirements need a standard format: Linus' feedback to "introduce a standard format" has been adapted. Our chosen standard to follow for all FRs is *user stories.* Thus, all functional requirements have been reworded into the format:

"As a <role>, I want <requirement> (so that <benefit>)."

As for quality requirements, we chose to go with the MoSCoW method proposed by Linus in his feedback. In this case, all requirements are given a different level of priority by the use of "must", "should", "could" and "won't" in each of their descriptions.

Reduce Scope of Project: Linus' feedback prompted us to reduce the number of functional requirements we were promising. Firstly, we clearly indicated bonus requirements F5 and F6 as such with explicit use of the "[Bonus]" tag. Additionally, we decided to remove FR7 ("Data Comparisons") as the inclusion of the two bonus FRs already extends the project's requirements.

Reduce Number of QRs: We have chosen to remove and reformat our QRs in the following ways:

- We identified the quality requirement **QR3** ("Extendable Interface") as redundant, given that it deals with the extendability of our system, just as **QR2** ("Extendable Metrics") does. Thus, we decided to merge both into one QR2.
- The quality requirement **QR4** ("Meaningful Colors") has been removed completely from the list of quality requirements. Instead, a new functional requirement has been added to tackle the interactivity aspect of our system. We identified that our previous F4 for "Data Representation" deals with similar representation-related components of our system, thus, we have re-labeled F4 to F4a and assigned our newly created requirement ("Interactive Interface") to F4b.
- The quality requirements **QR5** ("Help Page") and **QR6** ("Error Message") have been merged into a newly created QR4 as both aim to provide feedback on possible actions the user may take.
- We have made the decision to remove **QR8** ("Data Protection") because our system does not have a main focus on security.
- We have removed **QR7** ("Response Time") after determining that promising a "fast" response time cannot be guaranteed. Limitations on the speed of extracting GitHub data added uncertainty to this promise of ours.

Reduce ambiguity of FRs: We changed the wording of our functional requirements to clarify any vague descriptions:

- To make our functional requirement F3 more specific, we have clarified what we mean by **"relevant metrics"**. Relevant metrics that we have identified are the following: number of lines added/deleted, number of files modified, number of merges, top collaborators, and statistics about which days had the most commits.
- The format for the **written report** in F5 has been made more explicit. It is now specified that the written report will be using an A4 PDF format.

Introduce a reason for FRs: This feedback point is covered by our use of user stories to express function requirements by the "<benefit>" description.

2 Class Diagram

Author: Lennart

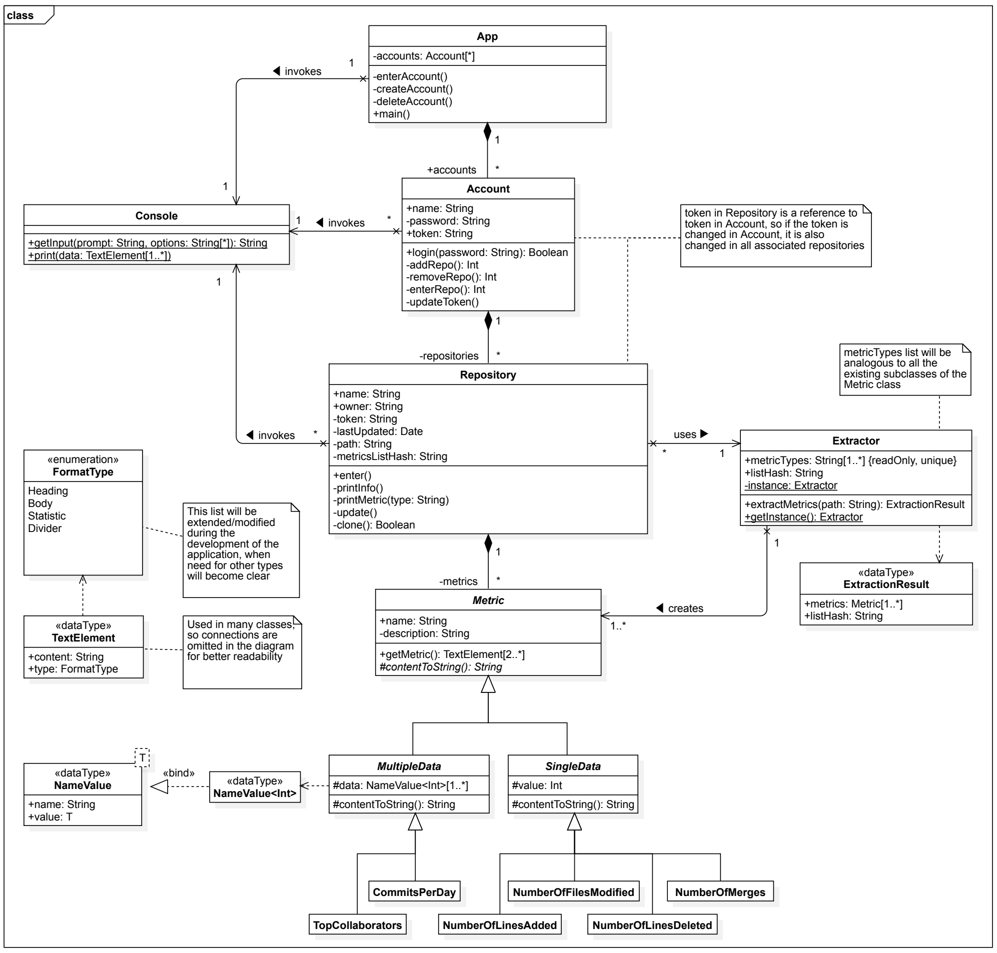


Figure 1: Class Diagram

Note: we omitted any getter and setter methods in this diagram to keep the focus on the inter-class structure and avoid polluting the diagram with unnecessary information. It can be expected that all the appropriate getters and setters will be present in the implementation. Furthermore, some functions that are only used for class-internal functionality are omitted for the same reasons.

2.1 Data Types

TextElement: This data type is used to provide the printing utility (here the **Console**) with both content to be printed as well as the desired formatting of the printed content. The format of the content is specified as an option from the enumeration **FormatType**. The list of format types is likely to be expanded during the development of the application, as the need for more types might arise. Even though printing to a command line output provides only little option for formatting, including the *formatType* for every instance of **TextElement** facilitates customization of the output styling at any point in the development.

NameValue<T>: As objects of the type **MultipleMetrics** need to store data in the form of pairs with a name and a value, this data type combines a *name*, represented as a String, and a *value* which can be of any type. An example use would be the storing of the number of commits per day, where the number of commits is the *value* and the name of the day is the *name*.

ExtractionResult: When metrics are extracted, the **Extractor** returns a hash of the list of metrics that were used along with the data itself. This enables quick evaluation of whether existing metrics of some repositories are up-to-date with the current list of available metrics, which is needed, as the list of metrics is extendable and might change over time. In order to return both the extracted data and the hash, this data type provides such a container. It is to note here, that this data type is almost identical in functionality to **NameValue**, but we decided to still include it for better clarity in the system, as the name of **NameValue** is not fully accurate for this use.

2.2 Static Classes

Console: As it is important to provide a common interface for the input and output (I/O) of the system in order to enable a possible modification of the desired I/O channels at any point in the development and offer an abstraction for the implementation details of I/O operations, the **Console** class provides two static functions for I/O.

- *getInput()* prints a prompt to the user and returns the input it receives back to the caller. Additionally to the prompt, this function also takes an array of strings as an argument which represents the possible options the user can choose. These options can be used to offer auto-completion to the user.
- *print()* takes an array of **TextElement** objects and prints the content of each in the specified format.

2.3 Singleton Classes

Extractor: An instance of the **Extractor** is created when the application is started and is responsible for loading all available metrics, storing their names in a set, and saving a hash of the list for the above-explained reasons. When the metrics of a repository are to be extracted, the *extractMetrics()* takes the path of where the repository was cloned to and returns an object of **ExtractionResult**, which groups the extracted metrics with the hash of the list of metrics that was used.

2.4 Regular Classes

App: The main entry point into the program for the user is given by the **Application** class. An instance of it is created when the program is started and contains all the Accounts that are present (either created at runtime or loaded from a save in the beginning). The *main()* method is an interaction loop in which the user can create, delete, or log into an account or quit the program. All communication with the user is done through the **Console**.

Account: As the application supports multiple users and offers the preservation of cloned/extracted repositories between application uses, the **Account** class stores all information and data related to a specific user. It furthermore offers a simple layer of protection by only allowing access to the class' attributes and methods after providing authorization by means of a valid password. This authorization is done by the *login()* method, which provides the entry point into an account. When the password was provided correctly, the method starts the interaction loop of the account and only returns back to the calling app when the user selected to quit the account. The return value of *login()* represents whether the password was correct (and thereby the account was active for some time) or the password was incorrect. All other methods of the **Account** class are for the functionality the class offers to the user (like adding a repository, entering a repository, and deleting a repository). For communication information to and from the application user, the class uses the **Console**.

Repository: An instance of **Account** can have any number of repositories. All the attributes and methods relevant to the repository are therefore grouped in the **Repository** class. When a new **Repository** instance is created by an **account**, it is provided with an *owner*, *name*, and *token* from which it can construct the proper address for cloning the GitHub repository into a folder for which it saves the location in *path*. When the cloning is unsuccessful, the constructor will throw an exception which is to be handled by the caller. The *enter()* method provides the entry point into the interaction loop of a **repository** similar to the *login()* method on the **Account** class. The **Repository** class uses the **Console** for information exchange with the user in the same way as the **Account** class. It furthermore uses the instance of the **Extractor** to get its metrics. The aforementioned metrics are saved in the *metrics* array with the corresponding hash in *metricsListHash*. Additionally, the time of the last update (i.e., pull) is saved as well.

Metric: Instances of sub-classes of the abstract class **Metric** are created by the instance of the **Extractor** and belong to exactly one **repository**. The class provides only the *getMetric()* method, which uses the *name* and *description* of the metric in combination with the *contentToString()* method to construct an array of **TextElement** objects that can then be sent to the **Console** for printing.

MultipleData & SingleData: These classes represent the two different types of metrics. **SingleData** are metrics that only contain a value (e.g., NumberOfMerges), while **MultipleData** are metrics that contain multiple **NameValue** pairs (e.g., CommitsPerDay). Both of these abstract classes implement the *contentToString()* method in the way that is appropriate for the type of data (single vs multiple data).

3 Object Diagram

Author: Laura and Dovydas

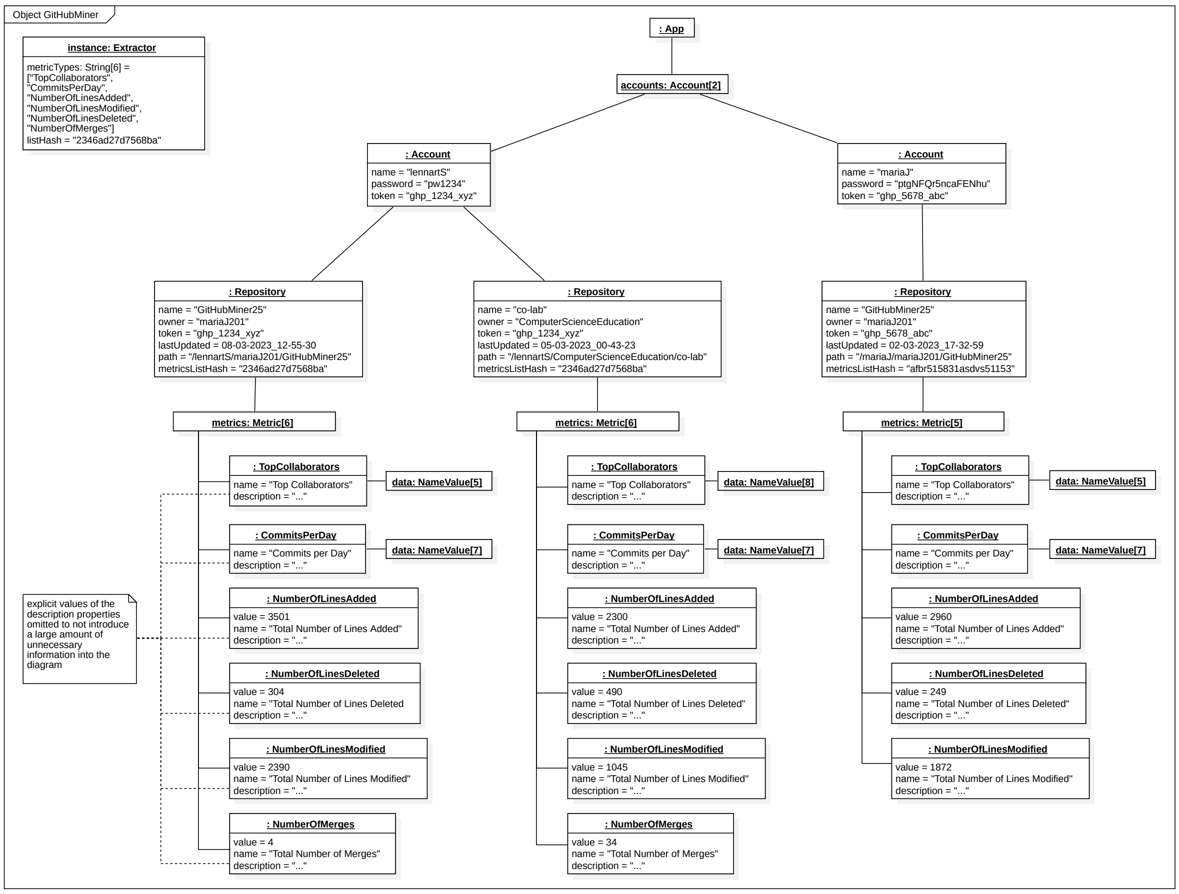


Figure 2: Object Diagram

The object diagram shown in Figure 2 demonstrates a single snapshot of our system in which two accounts exist, both with independent repository instances. **Account** object with *name* = "lennartS" has added two repositories to track, namely "GitHubMiner25" and "co-lab". The other **Account** object with *name* = "mariaJ" has only added the "GitHubMiner25" repository to its collection.

Each of the **Repository** objects contains a list of extracted metrics with their values and descriptions. The metrics have two different representations: metrics belonging to the **SingleData** class have an integer value, while **MultipleData** classes have NameValue pairs (for instance, **Repository** object with *name* = "co-lab" has **NumberOfLinesAdded** object with value *value* = 2300 and **CommitsPerDay** object with seven nameValues representing number of commits for each day of the week).

Even though both accounts are connected to the same **Repository** with *name* = "GithubMiner25", **Account** with *name* = "MariaJ" has an earlier *lastUpdated* date than that of **Account** with *name* = "lennartS". This difference in dates has a notable effect on two aspects of each **Repository** object: the values of existing Metrics and the number of total Metrics.

Firstly, the more recently updated repository has differences in the *Metric* values. For example, **NumberOfLinesAdded** object has *value* = 3501 in the more recently updated repository, while the same metric has a lower value of 2960 in the **Repository** with *lastUpdated* = 02-03-2023_17-32-59.

Secondly, since the interface was extended between these two dates with new *metric* "NumberOfMerges", the "GithubMiner25" **repository** in "mariaJ" **Account** does not have this newly added metric. This distinction can be seen in the *metricsListHash* values that differ between repositories. Additionally, the **Extractor** object instance specifies the existence of six *metricTypes* with a *listHash* equal to the *metricsListHash* in the more recently updated **Repository** object.

4 State Machine Diagram

4.1 Repository Class

Author: Lennart

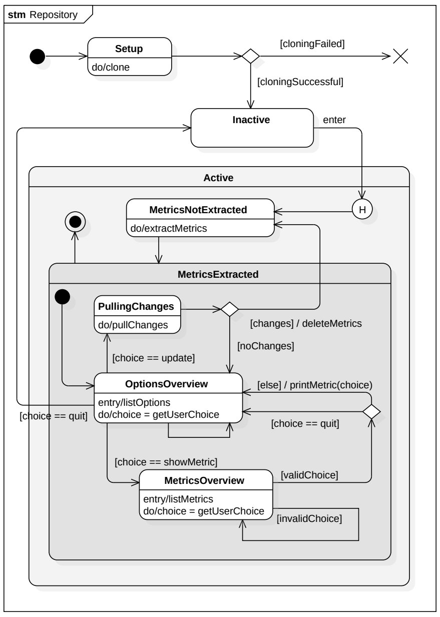


Figure 3: State Machine Diagram of the Repository Class

When an object of the **Repository** class is first created, it enters the ***Setup*** state. In this state, the object tries to clone the repo (name, owner, and token are given to the constructor). If this operation fails (as the given details might be wrong) the object reaches the terminate node and seizes to exist - a new object needs to be created to try again. If the cloning was successful, the object reaches the ***Inactive*** state, in which it remains until it is activated by the *enter* method.

When the object is activated, it enters the history pseudo state in the composite ***Active*** state. If the object was active before (and the metrics were extracted), the object will enter the ***MetricsExtracted*** state. If the object was not active before (or the metrics were not yet extracted), the object enters the ***MetricsNotExtracted*** state, in which it starts extracting the metrics and remains until the operation is done upon which the object will transition to the ***MetricsExtracted*** state.

When the ***MetricsExtracted*** composite state is entered, the ***OptionsOverview*** state becomes active. On enter, this state lists the available options and then waits for the user to enter a choice. Depending on the choice, the object transitions to different states:

update: The object transitions to ***PullingChanges***, where changes in the repository are pulled. After this activity is done, there are two possible cases:

1. no changes in the repository since the last update
2. at least one change in the repository since the last update

In the first case, the object transitions back to the ***OptionsOverview*** state. In the second case, the existing metrics are deleted and the object returns to the ***MetricsNotExtracted*** state, where it starts extracting the metrics from the new data.

showMetric: The object transitions to ***MetricsOverview***, where a list of available metrics are shown. The object remains in this state until the user has entered a choice. If the choice is invalid, the object reenters the same state and waits for a new choice. If the choice is valid, there are two possible cases:

1. the user selected quit
2. the user selected a metric

in both of the cases, the object transitions back to ***OptionsOverview*** but in the second case, the object also prints the metric data corresponding to the user's choice.

quit: The object transitions back to ***Inactive***. At this point, the caller that activated the repository object reclaims the application flow. The object remains in the ***Inactive*** state until it is activated again by the *enter* method.

If the choice of the user is invalid (i.e., none of the above-mentioned), the object re-enters ***OptionsOverview*** and thereby lists the available options again.

4.2 Account Class

Author: Dovydas

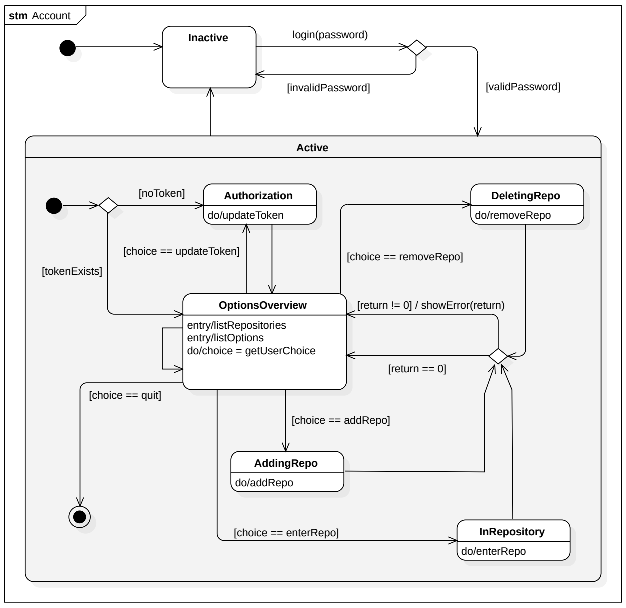


Figure 4: State Machine Diagram of the Account Class

When an object of the **Account** class is created, it enters the ***Inactive*** state. The object stays in this state until the login event with a particular password is called. If the password is valid, **Account** enters the composite ***Active*** state, otherwise, it returns to the ***Inactive*** state.

Once the **Account** class reaches the composite ***Active*** state, a check whether a token is connected to the **Account** is performed. If the token exists, the **Account** moves to the ***OptionsOverview*** state. Otherwise, if there is no token associated with the current class, before moving to the ***OptionsOverview*** state, **Account** enters the ***Authorization*** state where the updateToken activity prompts the user to enter a token for the **Account** object.

In the ***OptionsOverview*** state two entry activities are performed: the list of repositories connected to the account and the possible interaction options are displayed to the user. Then, the **Account** class waits in the ***OptionsOverview*** state until user has entered an interaction choice. The possible interaction choices are:

1. Updating the token. In this case the **Account** class enters ***Authorization*** state where user enters the updated token.
2. Adding a repository to the repository list. Here the **Account** class enters ***AddingRepo*** state where user is prompted to enter the details (owner name and the name of the repository) for the creation of a new Repository class.
3. Deleting a repository from the repository list. For that the **Account** class enters ***RemoveRepo*** state where the user is prompted to select a repository for its removal.
4. Entering a repository. The **Account** class transitions to ***InRepository*** state where the user selects a repository from the given list. The **Account** class stays in the ***InRepository*** state until the interaction within selected Repository class ends.
5. Exiting the Account. In this case, the **Account** class is transferred to the final state of the composite ***Active*** state, from where it returns to the ***Inactive*** state.

If the user enters an invalid choice, a self transition is performed in ***OptionsOverview*** state, after which the list of repositories and interaction options is displayed again to the user. After the internal activities of the first four interactions (updating token, adding/removing/entering a repository) are finished, the **Account** class returns back to the ***OptionsOverview*** state where user can select another interaction choice. In a case when a repository related interaction (addition, removal, entering of a repository) returns an non-zero return code, a related error code is displayed to the user during a transition to ***OptionsOverview*** state.

5 Sequence Diagram

5.1 Account Options

Author: Maria

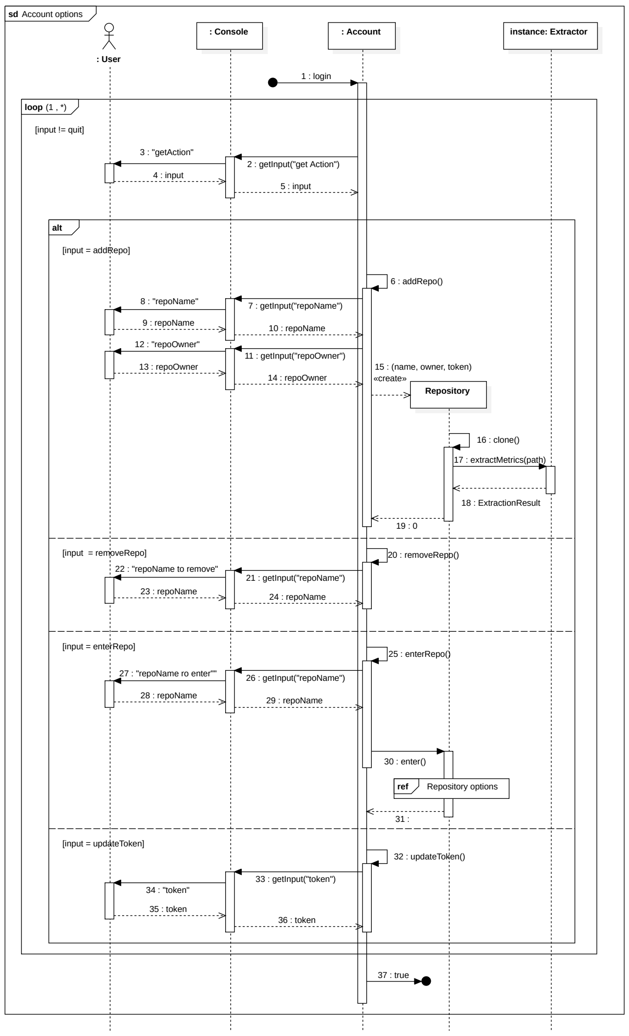


Figure 5: Sequence Diagram of the Account Options

The sequence diagram depicted in Figure 5 shows the communication sequence between a User and the instances of the classes **Console**, **Account**, and **Extractor** when the main loop from the **Account** class asks the user for the action they want to perform, after having selected and successfully logged-in in an account. The interaction starts with the `message login` being sent to **<u>Account</u>**, in this diagram it is not relevant to express its source and thus a found message is used. This message activates the main loop of the **Account** class, where the User is prompted to enter their next action. This triggers the interaction **<u>Console</u>** - **<u>User</u>** by the *getInput()* method; its result is stored in the input variable. From this point there are five possible execution paths: add a repo, remove a repo, enter a repo, update token, and quit from the account(exiting the main loop in **Account**). When this last option is chosen by the user, the loop is exited and *login()* returns true to a receiver that is not relevant(as the sender that initiates this sequences of actions); this is depicted in `message 37`. The remaining four options the User can choose from are represented by an alt fragment.

The first operand represents the alternative path for when input is addRepo. This triggers the method *addRepo()*, which needs to collect the repoName and repoOwner and does so through *getInput()* (`message 7-14`). Assuming the token has already been provided by the User, **<u>Account</u>** creates a new instance of **Repository** (`message 15`) passing as the arguments to the constructor of this class the token, repoName, and repoOwner. The constructor also clones (`message 16`), which in turn signals with `message 17` the **<u>Extractor</u>** to *extractMetrics()* to the specified path. The response `message 18` from **<u>Extractor</u>** to **<u>Repository</u>**

regarding the extracted metrics is of type ExtractionResult and with this the constructor finalizes its activities, returning the success code 0 to **Account**(`message 19`).

The second operand shows the messages exchanged when the input is removeRepo. In this case, *removeRepo()*(`message 20`) asks the User for the name of the repo to remove with the help of *getInput()* (`message 21-24`). The third operand is entered when the user inputs enterRepo. The method *enterRepo()* from the **Account** object questions the User the repoName to enter(`message 26-29`). In short, after the information needed is collected, `message 25` enters the repo and starts the main loop of **Repository**, *enter()*(`message 30`). Given the complex interaction sequences of the *enter()* main loop, an interaction reference Repository options has been included and modeled separately. Whenever this loop is exited, the void reply `message 31` is communicated back to **Account**, so that this instance can now proceed as well with its main loop(*login()*). Finally, the fourth operand of the alternative interaction is active when the User enters as input updateToken(e.g. for when the token has expired). For this path in execution, through `message 32` *updateToken()* gets the input from the user(`message 33-36`) and updates the information in the **Account**'s *token*.

It is worth noting that all the messages from the **Console** to the User are an indication for the content to be printed (not necessarily what will be exactly used in the implementation) and all error handling (e.g. invalid repo details) are not presented in this diagram. In addition, all messages aforementioned are synchronous given that the sender must wait until receiving the response message to continue with the proceeding activities.

5.2 Repository Options

Author: Maria

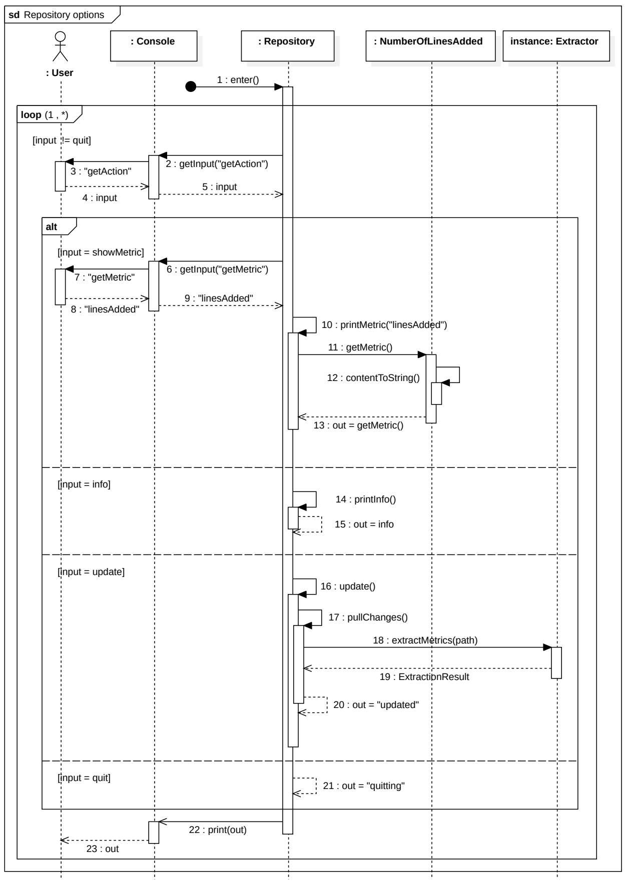


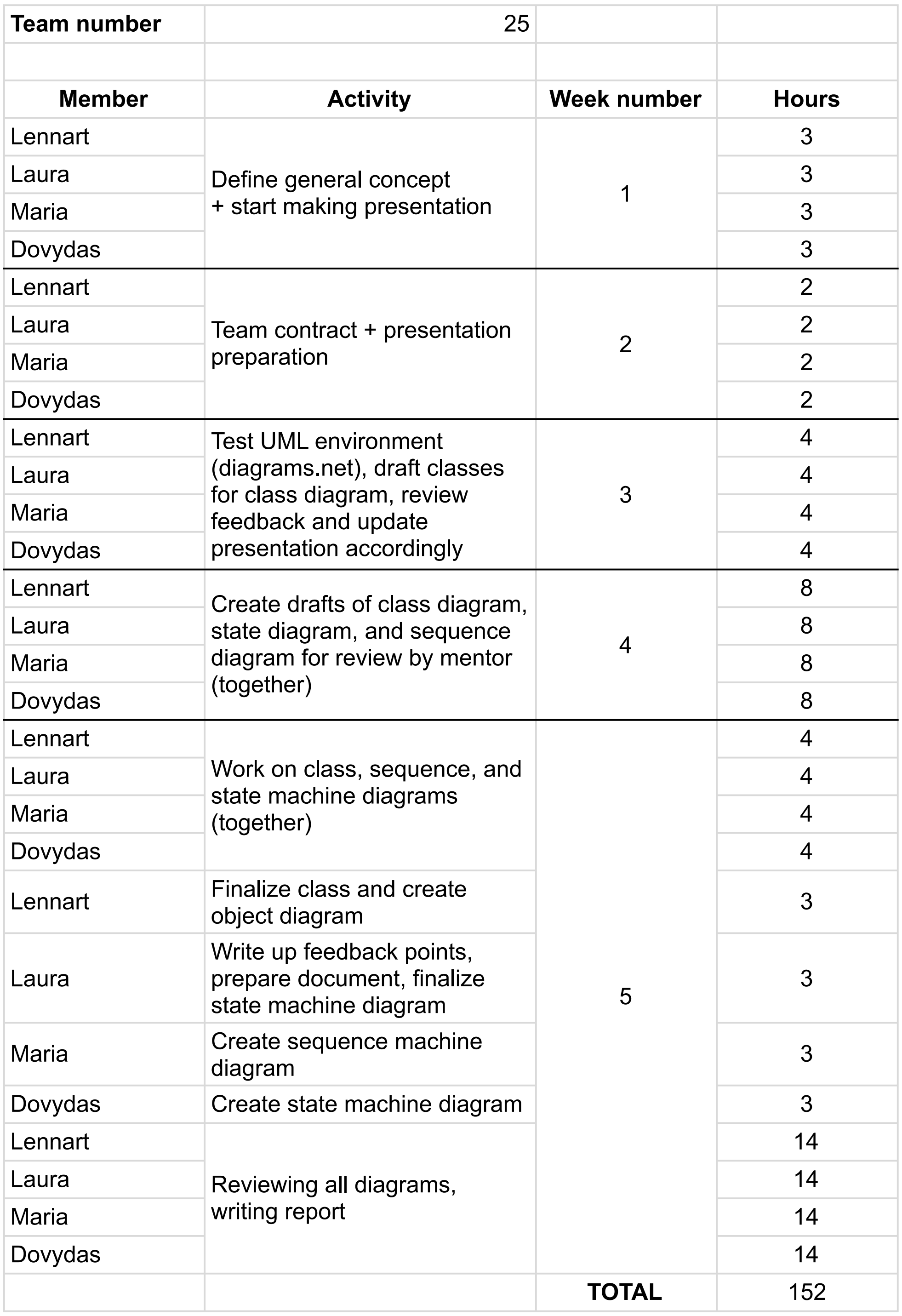
Figure 6: Sequence Diagram of the Repository Options

The “Repository Options” sequence diagram from 6 shows the interactions between a User, a **Console**, a **Repository**, **NumerOfLinesAdded**, and an **Extractor** when the main loop of the **Repository** class, *enter()*, is running. Note that all messages that follow are synchronous unless stated otherwise. As the sender of the first message (*enter()*) is not relevant in this situation, it is modeled as a found message. Immediately afterwards, the main loop in **Repository** starts iterating to retrieve the user selection; its main body gets the user input choice by getting the user input (`message 2-5`). An alternative interaction shows the four different options that can be chosen by the user from this point in time: show metrics, get info menu, update repo or quitting. The aim with the last message from each operand(`message 13, 15, 20, 21`) is to set the value of the variable out so that there is a single asynchronous `message print(out)` from the **Repository** to the **Console** (`message 22,23`) instead of having repeated instances of this messages on every operand.

For the first case, when the user inputs showMetric, **Repository** asks the User for the type of metric they are interested in(`message 6,7`). Taking as an example, the User replies with “linesAdded”, thus **Repository** starts the process of printing the metric(`message 10`), which requires the interaction with **NumberOfLinesAdded**, by *getMetric()* (`message 11`). **NumberOfLinesAdded** transforms the *contentToString()*(`message 12`) and out is set to the result of *getMetric()*. For the second operand where the user inputs info, *printInfo()*(`message 14`) sets the variable out to the info(`message 15`). For the third possibility the user inputs update, which by means of `message 16` triggers *update()*. This interaction needs *pullChanges()*(`message 17`), which once again extracts metrics through `message 18` that gets a reply from **Extractor** with the ExtractionResult(`message 19`). After returning to **Repository** out is set to “updated”. Finally, the last operand covers the case when the user inputs quit, to which the only interaction happening is setting the variable out to “quitting”(`message 21`) so that it can be printed by the **Console**.

It is worth noting that `message 8` and `message 9` are shorter versions of the actual string the user would type and are omitted for clarity reasons.

6 Time Logs



Team number	25		
Member	**Activity**	**Week number**	**Hours**
Lennart	Define general concept + start making presentation	1	3
Laura			3
Maria			3
Dovydas			3
Lennart	Team contract + presentation preparation	2	2
Laura			2
Maria			2
Dovydas			2
Lennart	Test UML environment (diagrams.net), draft classes for class diagram, review feedback and update presentation accordingly	3	4
Laura			4
Maria			4
Dovydas			4
Lennart	Create drafts of class diagram, state diagram, and sequence diagram for review by mentor (together)	4	8
Laura			8
Maria			8
Dovydas			8
Lennart	Work on class, sequence, and state machine diagrams (together)	5	4
Laura			4
Maria			4
Dovydas			4
Lennart	Finalize class and create object diagram		3
Laura	Write up feedback points, prepare document, finalize state machine diagram		3
Maria	Create sequence machine diagram		3
Dovydas	Create state machine diagram		3
Lennart	Reviewing all diagrams, writing report		14
Laura			14
Maria			14
Dovydas			14
		TOTAL	152

Figure 7: Time Log up until Week 5