Assignment 2

Team number: Team 61 - RepoRangers
Team members

Name	Student number	Email
Ewa Kaleta	2745573	e.k.kaleta@student.vu.nl
Naomi Maronic	2740042	n.maronic@student.vu.nl
Vincent Kohm	2726735	v.n.kohm@student.vu.nl
Zohaib Zaheer	2735075	z.zaheer@student.vu.nl

Do NOT describe the obvious in the report (e.g., we know what a `name` or `id` attribute means), focus more on the **key design decisions** and their "**why**", the pros and cons of possible **alternative designs**, etc.

Format: establish formatting conventions when describing your models in this document. For example, you style the name of each class in bold, whereas the attributes, operations, and associations as underlined text, objects are in italic, etc.

Summary of changes from Assignment 1

Author(s): Ewa Kaleta, Vincent Kohm, Naomi Maronic

Link to the project pitch

Provide a bullet list summarizing all the changes you performed in Assignment 1 for addressing our feedback.

- Overview:
  - Specification of stakeholder / type of users, separate from system description.
  - Condensed the system description into a high-level overview rather than a detailed description.
- Features
  - Categorising features using MoSCoW - introducing the reason for each feature (why users might need this, why it is essential for the system, etc.)
  - Breaking down features into smaller ones and adding missing ones + redistribution of features
- Quality requirements
  - Deletion of previous QR1, 2 & 7
  - Change of quality attributes
- Team Contract

- Elaborated on how we define “*work equitably”:*
    - This means that before we tackle an assignment or a task we discuss how we could divide this into equal parts and assign them to individual team members. Once the task is assigned we expect each member to attempt it and present some effort to the team. If one team member is having trouble completing their task we expect them to communicate this with the team so that we can collectively approach it and solve it before the deadline. We believe that with this approach each team member will have contributed equally over the duration of the project.
- Expanded on disputes:
    - The team will take action against freeriders is:
        - that individual repeatedly fails to deliver results
        - that individual repeatedly fails to attend a meeting (i.e. TA meetings or team meetings)
        - that individual repeatedly fails to deliver their task without any communication and/or excuse
- Expanded on Team Goals:
    - The team goals are to dive deeper into git and GitHub and gain a wider knowledge of its functionalities and the corresponding commands. This is why we collectively choose to go forward with this project. In addition, we are hoping to grasp a better understanding of common design patterns and apply our knowledge in the assignments

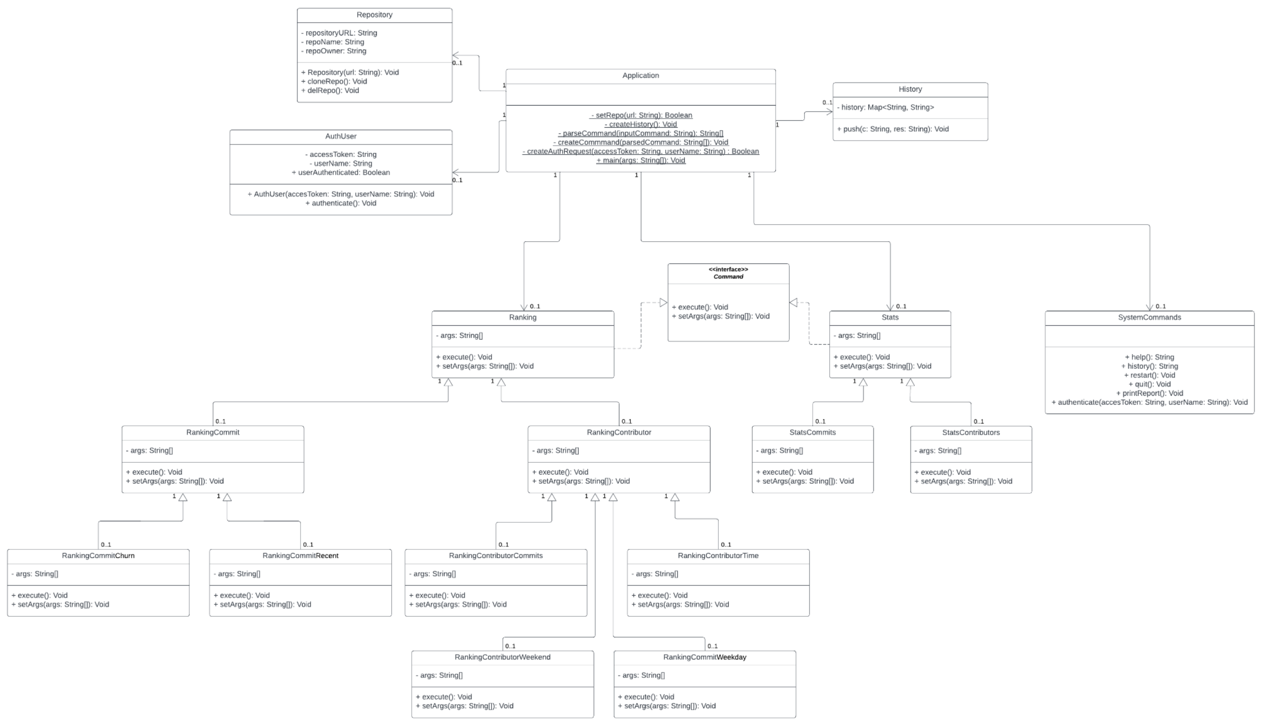
Maximum number of pages for this section: 1

Class diagram

Author(s): Vincent Kohm, Zohaib Zaheer

Link to the Class diagram

This chapter contains the specification of the UML class diagram of your system, together with a textual description of all its elements.


Repository
- repositoryURL: String
- repoName: String
- repoOwner: String
+ Repository(url: String): Void
+ cloneRepo(): Void
+ delRepo(): Void
Application
- setRepo(url: String): Boolean
- createHistory(): Void
- parseCommand(inputCommand: String): String[]
- createCommmand(parsedCommand: String[]): Void
- createAuthRequest(accessToken: String, userName: String) : Boolean
+ main(args: String[]): Void
History
- history: Map<String, String>
+ push(c: String, res: String): Void
AuthUser
- accessToken: String
- userName: String
+ userAuthenticated: Boolean
+ AuthUser(accesToken: String, userName: String): Void
+ authenticate(): Void
<<interface>>
Command
+ execute(): Void
+ setArgs(args: String[]): Void
Ranking
- args: String[]
+ execute(): Void
+ setArgs(args: String[]): Void
Stats
- args: String[]
+ execute(): Void
+ setArgs(args: String[]): Void
SystemCommands
+ help(): String
+ history(): String
+ restart(): Void
+ quit(): Void
+ printReport(): Void
+ authenticate(accesToken: String, userName: String): Void
RankingCommit
RankingContributor
StatsCommits
StatsContributors
RankingCommitChurn
RankingCommitRecent
RankingContributorCommits
RankingContributorTime
RankingContributorWeekend
RankingCommitWeekday
- args: String[]
+ execute(): Void
+ setArgs(args: String[]): Void
1
0..1

Application: This class is the central class that all other classes as well as the user interacts with – it has instances of the *Repository, AuthUser, History, Ranking, Stats,* and *SystemCommands* classes therefore it has one way associations with all of these classes. It serves as an invoker class that creates objects for all the other classes that are required by the system to function. The user never directly interacts with any other class but *Application*. All of the methods in *Application* are static as no other class calls these methods – they are only used to create instances of the connected classes. The most important of these methods are as follows:

- *main*: This is the method where our application is initiated from. This is the method that is "run" on startup, it asks the user for the URL of the repo to clone, the command to run, and it runs all the other methods of the Application class.
- *createAuthRequest*: If the user wishes to authenticate themselves in order to work with private repos, *main* runs the *createAuthRequest* method. This method works by creating an instance of the *AuthUser* class and running the *authenticate* method from that class on this instance.
- *setRepo*: This is the only method that needs to be run for the program to continue as without it there is no repository to run commands on. This method creates an

instance of *Repository* and calls *cloneRepo* (from *Repository)* on this instance to clone the repo into our local machine.
- *createCommand*: This method takes an array of strings, evaluates which of the *Command* classes is associated with the command, creates the associated instance of that *Command* class, and calls the *execute* method.
- *createHistory*: Creates an instance of *History* which is used to store the command and result history for the *printReport* command.

Repository: This class is responsible for storing information about the repository the user wishes to clone to the local machine. As such, its attributes store basic repository information which is useful for printing the information in the report. This repository has a constructor method which is used to set the attributes. Its *cloneRepo* method (as explained in the previous paragraph), clones the repo using *git clone* via the Java *ProcessBuilder* library. *delRepo* deletes the cloned repo from the local machine when *restart* or *quit* are called. *Repository* only has a one way association from *Application*.

AuthUser: This repository holds the authentication information for a user who wishes to work with private repositories. Its constructor is used to set the *accessToken* and *userName* attributes while its *authenticate* method sends a request to GitHub via a Java library (such as Eclipse Git or Kohsuke) using the access token provided. This method sets the *userAuthenticated* attribute to *True* upon successful authentication and *False* upon unsuccessful authentication. As an extra security measure, the username entered by the user is checked against the username retrieved by the GitHub API client and only if those two match is the user allowed to work with private repos. *AuthUser* only has a one way association from *Application*.

History: History has an attribute with data type Map, a Java data structure, that stores the commands, as entered by the user, and the results, as produced by our system. Its only method, *push*, is used to push the command and its result to *history*. *History* only has a one way association from *Application*.

Command (interface): The command interface is simply a blueprint for all commands that interact with the repo have to follow. Its method *execute* is used to run the actual execution of commands using *git log*. Its other method, *setArgs*, sets any arguments provided by the user to the *args* attribute of the classes that implement *Command.* Its associations are of type *Implementation* and they can be traced to *Ranking* and *Stats* classes as they implement this interface (keep in mind that the subclasses of *Ranking* and *Stats* also implement *Command* but since the subclasses inherit from their parent classes, this relationship is not explicitly shown in the diagram).

Ranking: This class serves the command “ranking” and it prints a summary of all possible rankings. It is one of the classes that implements the *Command* interface, as such, it implements *execute* and *setArgs* (as described in the previous paragraph). It is initially instantiated and its methods are called by *Application* as such that is one of its associations. This class’ *execute* method is also responsible for creating an instance of one of its subclasses and calling this instance’s execute function if the arguments provided by the user (and stored in the *args* attribute) dictate it do so – if this happens, then *git log* will not be run in this class but rather in a subclass.

RankingCommit: This class serves the command “ranking” with the argument “commit” and prints rankings of commits by churn and recency. It uses generalization to inherit from *Ranking* as such it also implements the *Command* interface, which includes *setArgs* and *execute*. This class’ *execute* may also be used to create an instance of *RankingCommitChurn* if the arguments entered by the user dictate so – if this happens, then *git log* will not be run in this class but rather in a subclass.

RankingCommitChurn: This class serves the command “ranking” when it is entered with the arguments “commit” and “churn” and prints the ranking of commits by churn. It uses generalization to inherit from *RankingCommit* as such it also implements the *Command* interface. If an instance of this class is created, it means that the execution of *git log* must be done by the *execute* function of this class. This is done by checking if the *args* attribute is *null* (which it should be given that this is the end of the tree branch).

RankingCommitRecent: This class serves the command “ranking” when it is entered with the arguments “commit” and “recent” and prints the ranking of commits by which was committed most recently. It uses generalization to inherit from *RankingCommit* as such it also implements the *Command* interface. If an instance of this class is created, it means that the execution of *git log* must be done by the *execute* function of this class. This is done by checking if the *args* attribute is *null* (which it should be given that this is the end of the tree branch).

RankingContributor: This class serves the command “ranking” with the argument “contributor” and prints multiple rankings of contributors by different factors such as commits and time. It uses generalization to inherit from *Ranking* as such it also implements the *Command* interface. This class’ *execute* may also be used to create an instance of *RankingContributorCommits, RankingContributorWeekend, RankingContributorWeekday,* or *RankingContributorTime* if the arguments entered by the user dictate so – if this happens, then *git log* will not be run in this class but rather in one of the subclasses.

RankingContributorCommits: This class serves the command “ranking” when it is entered with the arguments “contributor” and “commits” and prints a ranking of contributors by commits. It uses generalization to inherit from *RankingContributor* as such it also implements the *Command* interface. If an instance of this class is created, it means that the execution of *git log* must be done by the *execute* function of this class. This is done by checking if the *args* attribute is *null* (which it should be given that this is the end of the tree branch).

RankingContributorWeekend: This class serves the command “ranking” when it is entered with the arguments “contributor” and “weekend” and prints a ranking of contributors by who worked the most on weekends. It uses generalization to inherit from *RankingContributor* as such it also implements the *Command* interface. If an instance of this class is created, it means that the execution of *git log* must be done by the *execute* function of this class. This is done by checking if the *args* attribute is *null* (which it should be given that this is the end of the tree branch).

RankingContributorWeekday: This class serves the command “ranking” when it is entered with the arguments “contributor” and “weekday” and prints a ranking of contributors by who was working the most on weekdays. It uses generalization to inherit from *RankingContributor* as such it also implements the *Command* interface. If an instance of this class is created, it means that the execution of *git log* must be done by the *execute* function of this class. This is done by checking if the *args* attribute is *null* (which it should be given that this is the end of the tree branch).

RankingContributorTime: This class serves the command “ranking” when it is entered with the arguments “contributor” and “time” and prints a ranking of contributors by who has been working on the repository the longest. It uses generalization to inherit from *RankingContributor* as such it also implements the *Command* interface. If an instance of this class is created, it means that the execution of *git log* must be done by the *execute* function of this class. This is done by checking if the *args* attribute is *null* (which it should be given that this is the end of the tree branch).

Stats: This class serves the command “stats" and it prints a summary of all possible stats from the cloned repository. It is one of the classes that implements the *Command* interface, as such, it implements *execute* and *setArgs* (as described in the *Command* paragraph). It is initially instantiated and its methods are called by *Application* as such that is one of its associations. This class’ *execute* method is also responsible for creating an instance of one of its subclasses and calling this instance’s execute function if the arguments provided by the user (and stored in the *args* attribute) dictate it do so – if this happens, then *git log* will not be run in this class but rather in a subclass.

StatsCommits: This class serves the command “ranking” when it is entered with the argument “commits” and prints stats related to the commits such as files changed, branch id etc. It uses generalization to inherit from *Stats* as such it also implements the *Command* interface. If an instance of this class is created, it means that the execution of *git log* must be done by the *execute* function of this class. This is done by checking if the *args* attribute is *null* (which it should be given that this is the end of the tree branch).

StatsContributors: This class serves the command “ranking” when it is entered with the argument “contributors” and prints stats related to the contributor. It uses generalization to inherit from *Stats* as such it also implements the *Command* interface. If an instance of this class is created, it means that the execution of *git log* must be done by the *execute* function of this class. This is done by checking if the *args* attribute is *null* (which it should be given that this is the end of the tree branch).

SystemCommands: This class does not implement the *Command* interface. This is because the *Command* interface is a blueprint for classes that implement repository commands and have arguments. Given that system commands do not have any arguments, it does not make sense for this class to use the *Command* interface. Therefore, this class’ only association is with *Application* which instantiates it and calls its methods when the user enters an associated command. Another difference between this class and all the previous *Command* classes is that this single class implements many commands whereas the previous classes each executed at most one command. Each of this class’ methods executes a single system command as laid out in our functional requirements. The methods have the same names as the commands and are as follows:

- *help*: print an ordered list of all possible commands the system supports with their uses for the user.
- *history*: print an ordered list of only all the commands the user has entered in the current session.
- *restart*: set the system back to the initialization stage (i.e. ask for a new repo), delete the currently cloned repo from the local machine, and clear history of commands and results.
- *quit*: exit the system, delete cloned repo from local machine, clear history of commands and results.
- *printReport*: output the history (which stores commands and results) to a text file.
- *authenticate*: same action as *authenticate* from *AuthUser* but allows users to authenticate themselves at any time during the session.

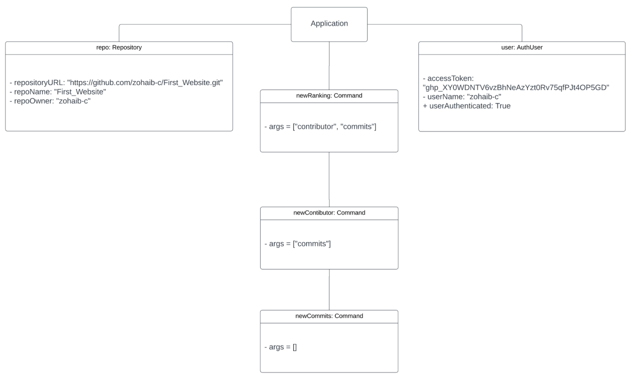
Object diagram

Author(s): Zohaib Zaheer, Vincent Kohm

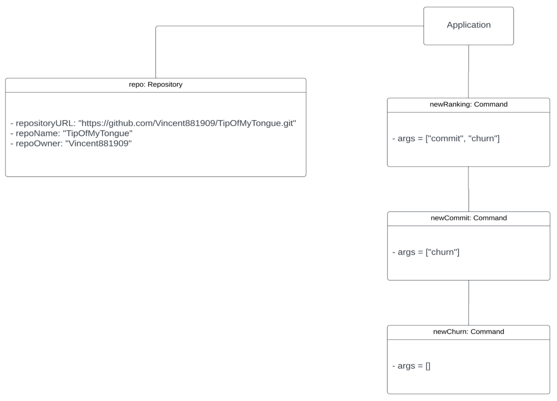
Link to Object Diagram 1
Link to Object Diagram 2

This chapter contains the description of a "snapshot" of the status of your system during its execution. This chapter is composed of a UML object diagram of your system, together with a textual description of its key elements.

Object Diagram 1


Application
repo: Repository
- repositoryURL: "https://github.com/zohaib-c/First_Website.git"
- repoName: "First_Website"
- repoOwner: "zohaib-c"
user: AuthUser
- accessToken: "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"
- userName: "zohaib-c"
+ userAuthenticated: True
newRanking: Command
- args = ["contributor", "commits"]
newContibutor: Command
- args = ["commits"]
newCommits: Command
- args = []

Object Diagram 2


Application
repo: Repository
- repositoryURL: "https://github.com/Vincent881909/TipOfMyTongue.git"
- repoName: "TipOfMyTongue"
- repoOwner: "Vincent881909"
newRanking: Command
- args = ["commit", "churn"]
newCommit: Command
- args = ["churn"]
newChurn: Command
- args = []

Description

Both snapshots exemplify the Application class running a command. In the first diagram the command is “ranking contributor commits” which ranks all contributors based on their commits. The second diagram runs the “ranking commit churn” command which prints the number of added and deleted lines and pathname and ranks them accordingly. Both diagrams have a Repository class initialised called repo. In diagram 1, the repository holds attributes of a private repository as opposed to diagram 2 in which the repo class holds attributes to a public repository. Besides the 2 different commands being illustrated, the first diagram also illustrates an authenticated user. Therefore, the AuthUser class is also initialised and named *user* in this case. In contrast, no authentication took place in the second diagram and thus no instance of the AuthUser class was created. The list of arguments provides our application with information about specific details required by the user. Therefore, one can observe that the list of arguments decreases as we move down in the command tree until there are no arguments left which means we have reached the lowest level in the tree and this specific command is triggered. We believe that these are valuable snapshots to present in order to gain a better understanding of how the different classes could be instantiated.

State machine diagrams

Author(s): Ewa Kaleta

Link to the State Machine Diagrams (see Page 1, Page 2 and Page 3 in the left corner for all the diagrams)

This chapter contains the specification of at least 2 UML state machines of your system, together with a textual description of all their elements. Also, remember that classes describing only data structures (e.g., Coordinate, Position) do not need to have an associated state machine since they can be seen as simple "data containers" without behaviour (they have only stateless objects).

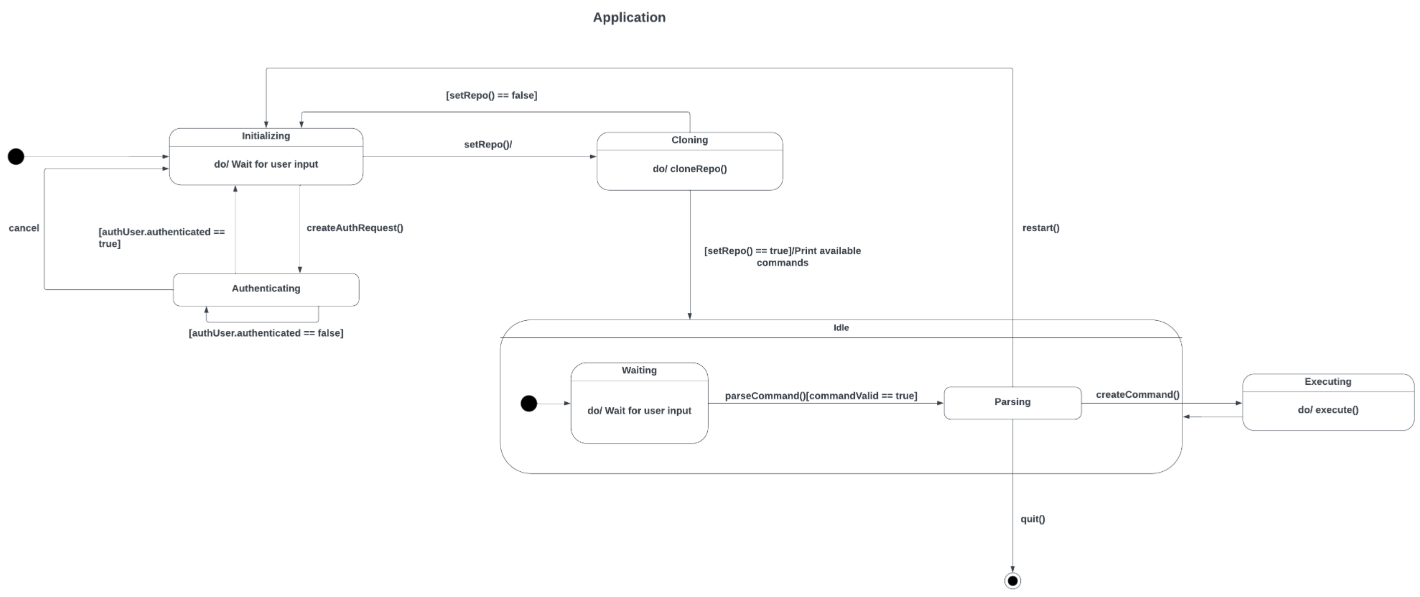
For each state machine you will provide:

- the name of the class for which you are representing the internal behavior;
- a figure representing the state machine;
- a textual description of all its states, transitions, activities, etc. in a narrative manner (you do not need to structure your description into tables). We expect 3-4 lines of text for describing trivial or very simple state machines (e.g., those with one to three states), whereas you will provide longer descriptions (e.g., ~500 words) when describing more complex state machines.

The goal of your state machine diagrams is both descriptive and prescriptive, so put the needed level of detail here, finding the right trade-off between understandability of the models and their precision.

Maximum number of pages for this section: 4

Class : Application



The initial state of the Application class is **Initializing**, where it waits for the user input. The user can either choose to authenticate their GitHub account to access private repositories (state transition to **Authenticating**) or skip this step and provide a URL for a public repository (state transition to **Cloning**).

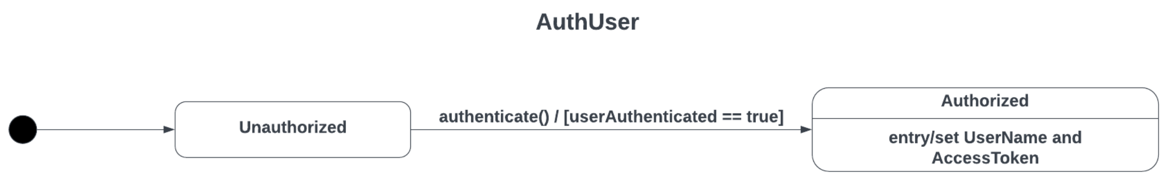
If the user chooses to authenticate, the *createAuthRequest()* function is called triggering state transition to **Authenticating**. From this point it is only possible to transition back to the **Initializing** state after the authentication succeeds, so the guard [*authUser.authenticated == true]* or if the user chooses to *cancel* ("please press c if you wish to cancel" - so this is a keyboard input rather than a function). If the authentication fails, guard [*authUser.authenticated == false],* the state doesn't change and the user can attempt authentication again.

The transition from **Initializing** to **Cloning** happens after the user inputs the repo URL and the *setRepo()* function is triggered. While in the "Cloning" state *cloneRepo()* is executed. Once done we check what *setRepo()* has returned. If the cloning fails, *setRepo() == false,* a transition back to the **Initializing** state occurs and the user can provide a different, correct, URL. Otherwise, if the cloning was successful and *setRepo() == true* (so it returned true) then a transition to **Idle** occurs. This transition is accompanied by the action *Print available commands***,** which displays all of the available commands with descriptions of what they do.

The **Idle** state is a composite state with two substates - **Waiting** and **Parsing**. After *setRepo()* evaluates to true transition to **Idle** happens followed by an immediate transition to the **Waiting** state. Here, the Application waits for user input - a command. Once a command is entered the *parseCommand()* is executed and if the guard [*commandValid]* is true a transition to **Parsing** occurs. This is to ensure that this transition only happens if the user inputs a correct command. If the command is invalid the user will be asked to enter a valid one and the state of Application will stay as **Waiting**.

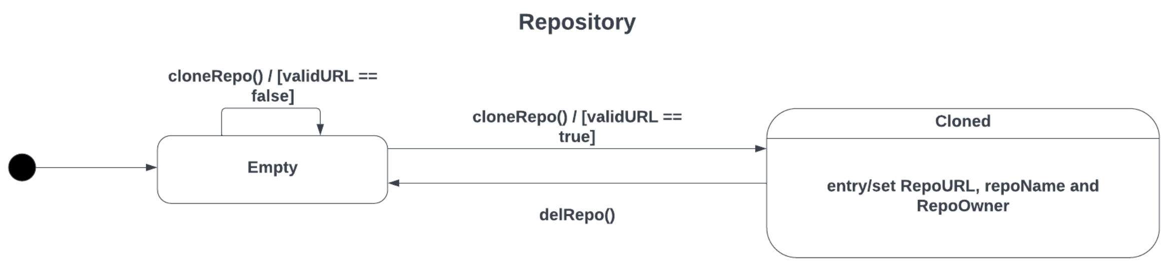
There are three different transitions that can happen from the **Parsing** state. If the parsed command is the *quit* command a transition to the final state occurs. If the parsed command is the *restart* command a transition back to the **Initializing** state occurs, where a new repo can be added. Otherwise, *createCommand()* is called and the Application exits the composite state **Idle** and transitions to **Executing**. In this state *execute()* is called and once it finishes the state transitions back to **Idle**.

Class: AuthUser


AuthUser
Unauthorized
authenticate() / [userAuthenticated == true]
Authorized
entry/set UserName and AccessToken

The AuthUser class has two states - **Unauthorized** and **Authorized**. The transition between the states can only be triggered once the *authenticate()* function is called and the guard *userAuthenticated* is true, so when the authentication succeeds. On entry to the **Authorized** state the UserName and Access Token are set. If the authentication fails the transition does not occur and AuthUser waits for another attempt (*authenticate()* call).

Class: Repository


Repository
cloneRepo() / [validURL == false]
Empty
cloneRepo() / [validURL == true]
delRepo()
Cloned
entry/set RepoURL, repoName and RepoOwner

The Repository class default state is **Empty -** when no Repository has been added (cloned) to the system yet. The transition to the **Cloned** state is triggered when the *cloneRepo()* function is called given the guard *validURL* evaluates to true. If it evaluates to false the transition does not occur. On entry to the **Cloned** state *RepoURL*, *repoName* and *RepoOwner* are set. Transition from the **Cloned** to the **Empty** state is triggered when *delRepo()* is called.

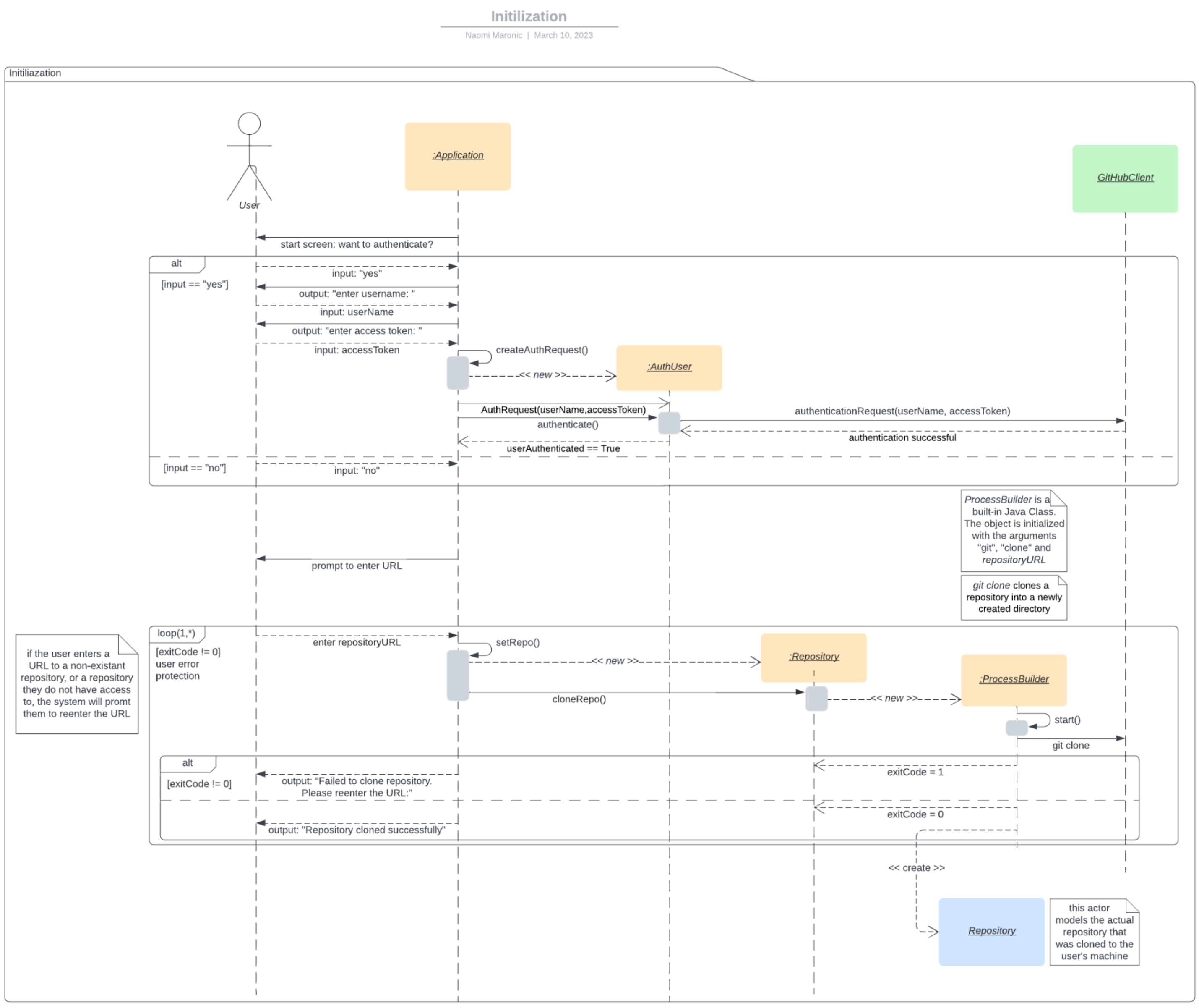
Sequence diagrams

Author: Naomi

We decided to use the sequence diagrams to model the initialization of the system, as well as one user interaction where a user wants to get some information about the contributors of the specified repository.
The initialization, which includes the authentication procedure as well as the cloning of the repository to the user's machine, is highly significant because of multiple reasons: first of all, during each interaction with the system, it is necessary to go through this initial process. Secondly, the cloning of the repository is a key part of the GitHub Miner. Without the cloned repository, no *git log* requests can be send and the user will not get any information about the repository. For this reason, we decided to model this interaction in a sequence diagram.
In addition, we thought it was important to show an example of the user entering at least one command and one system command. We chose to show *statsContributors* and end the interaction with quitting the GitHub Miner application.
As a result, we have modelled all important parts of the system, namely the initialization, the handling of commands and the exit.

Initialization of the GitHub Miner

Available here


Initilization
Naomi Maronic | March 10, 2023
Initiliazation
User
:Application
GithubClient
start screen: want to authenticate?
alt
[input == "yes"]
input: "yes"
output: "enter username: "
input: userName
output: "enter access token: "
input: accessToken
createAuthRequest()
<< new >>
:AuthUser
AuthRequest(userName,accessToken)
authenticate()
authenticationRequest(userName, accessToken)
authentication successful
userAuthenticated == True
[input == "no"]
input: "no"
ProcessBuilder is a built-in Java Class. The object is initialized with the arguments "git", "clone" and repositoryURL
git clone clones a repository into a newly created directory
prompt to enter URL
loop(1,*)
[exitCode != 0] user error protection
if the user enters a URL to a non-existant repository, or a repository they do not have access to, the system will promt them to reenter the URL
enter repositoryURL
setRepo()
<< new >>
:Repository
cloneRepo()
<< new >>
:ProcessBuilder
start()
git clone
alt
[exitCode != 0]
output: "Failed to clone repository. Please reenter the URL:"
exitCode = 1
exitCode = 0
output: "Repository cloned successfully"
<< create >>
Repository
this actor models the actual repository that was cloned to the user's machine

We assume that the user has started his GitHub Miner Program. The modelled interaction begins with the *Application* object printing the start screen to the Command Line Interface (CLI). The start screen includes a prompt asking the user whether they want to authenticate themselves with a GitHub access token. If the user wants to mine any of the private repositories they have access to, authentication is necessary. The log-in process is initiated after the user inputs “yes” in the CLI. The system, or more specifically the *Authentication* object, asks the user to input their GitHub user name and their access token. Afterwards, a call to the *createAuthRequest* function generates an authentication request and leads to an invocation of the *AuthUser* class. The *Application* object then initialises the *userName* and *accessToken* of the *AuthUser* object with *AuthRequest* function call. Next, the *authenticate* function is called and sends a request to the GitHub Client. The authenticate function returns the Boolean variable *userAuthenticated* which is set to true after a successful authentication response from the *GitHub Client*.

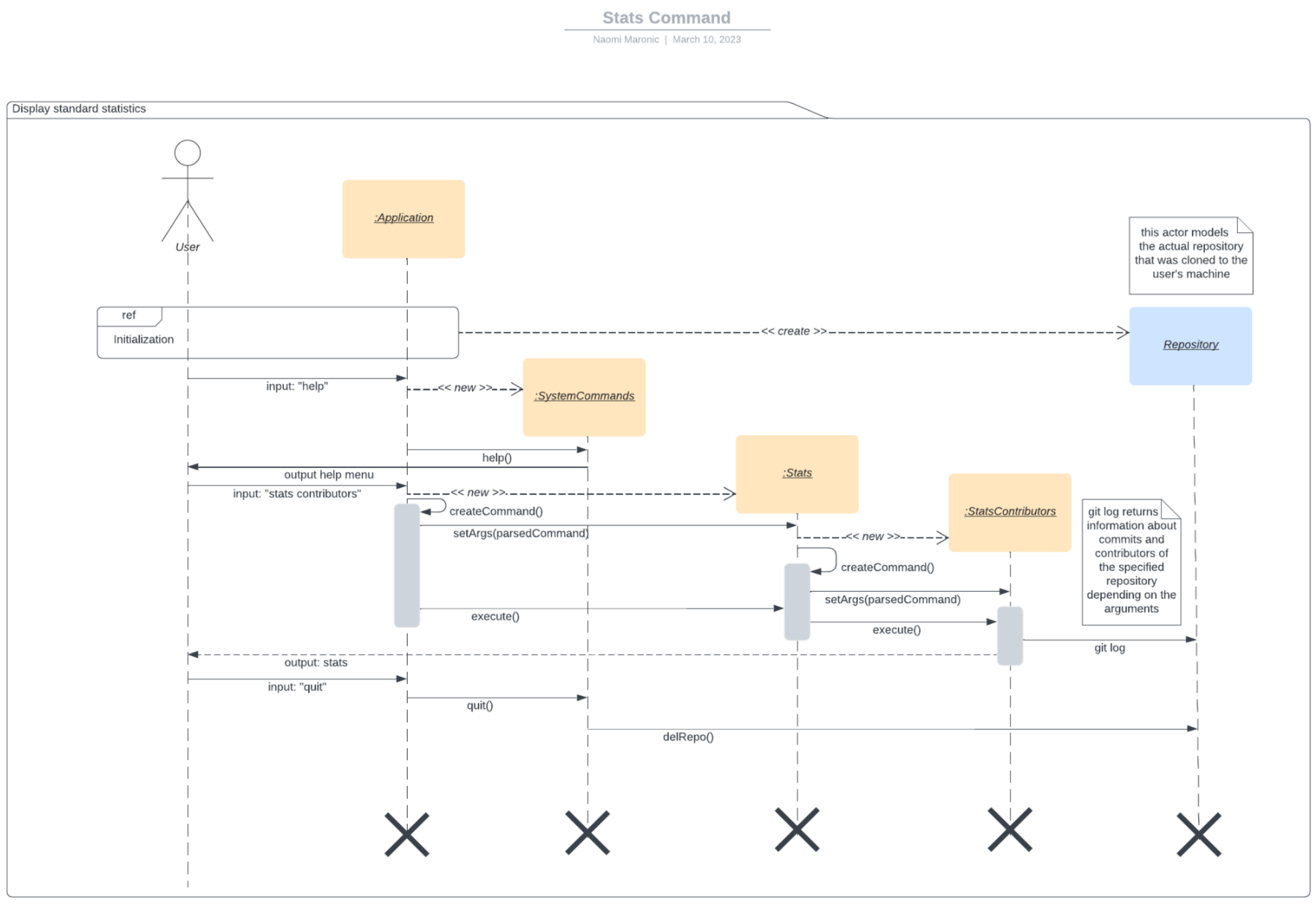
After authentication, or if the user skipped the log-in process by inputting “no” after the prompt, the interaction continues to ask the user to enter the URL to a repository on GitHub. In the following, a user-error-protection loop is entered that handles the input of the URL and the cloning process in general. If the cloning was not successful, because the URL is faulty or links to a repository the user does not have access to, the loop starts again. Once the cloning of the specified repository was successful, the loop is exited.

The loop starts with the user entering the *repositoryURL*. Afterwards, the *setRepo* function is called. In this function, a *Repository* object is instantiated and the *cloneRepo* function is invoked. Then, the *processBuilder* object is built which sends the *git clone* command to the *GitHubClient*. If the repository was successfully cloned from GitHub to the user's machine, the *exitCode* variable is initiated with 0. If the *exitCode* is not equal to 0, the repository either does not exist or the user is not granted access. The *Application* object informs the user through the CLI that the cloning failed. The loop starts again and, consequently, prompts the user to enter a valid URL. This is repeated until the *exitCode* is 0. That means the process builder successfully cloned the repository at the given URL to the user's machine. To inform the user about this, the *Application* object outputs to the CLI that the Repository has been cloned successfully. With this the initialization process is complete.

Display standard statistics

Our second sequence diagram models a user entering a command, namely *stats contributors*.

Available here



After the initialization process (which was modelled in the first sequence diagram) the user inputs “help” in the CLI to get more information about the commands and the workings of the GitHub Miner Program. As a result, the *Application* object instantiates the *SystemCommands* class and calls its *help* function, which then outputs the help menu. Afterwards, the user enters the “stats contributors” command. This leads to the creation of the *Stats* object. Still in the *Application* object, the *createCommand* function gets the *parsedCommand* as an argument and then enters the respective case of a switch-case statement. This creates the object of the matching subclass, in this case *StatsContributors*. The *setArgs* function, initialises the variables in the newly created object of the subclass, here *Stats*. The next part of the command (here: “contributors) is parsed which then leads to the instantiation of the *StatsContributors* object. The *setArgs* function parses the command again, but since the input string ends after “contributors” no new subclass gets instantiated. Both *setArgs* functions return. Next, *Application* calls the *execute* function, which is part of the *Command* interface. The most specific function, so the *execute* in the *statsContributors* object, is the one that gets executed. The *execute* function performs a *git log* request, with the respective arguments, so the request returns information about the contributors of the cloned repository. This information is then outputted on the CLI from the *StatsContributors* object.
Lastly, the user enters “quit” in the CLI to exit the application. This results in the *quit* function call to the *SystemCommands* object. As a result, the cloned repository is deleted with a call to *delRepo*, which is a command specified in the *SystemCommands* class. When the GitHub Miner application stops running, all objects get destroyed.

Time logs

Team Number	61 - RepoRangers		
Member	**Activity**	**Week number**	**Hours**
Zohaib Zaheer	Class Diagram	1+2	8
	Object Diagram	2	2
	Preliminary Implementation	2	2
	Assignment 2 Report	2	2
Naomi Maronic	Review of Assignment 1	1	2
	Sequence Diagram	1+2	10
	Assignment 2 Report	2	2
Vincent Kohm	Class Diagram	1+2	6
	Object Diagram	2	2
	Review of Assignment 1	2	1
	Assignment 2 Report	2	2
Ewa Kaleta	State Machine Diagram	1+2	10
	Review of Assignment 1	1	2
	Assignment 2 Report	2	2
Team Meetings	Tuesday 28th February	1	2
	Thursday 2nd March	1	2
	Friday 3rd March (TA)	1	1
	Tuesday 7th March	2	2
	Thursday 9th March	2	1
	Friday 10th March (TA)	2	1
		Total	62