Assignment 3

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

Summary of changes from Assignment 2

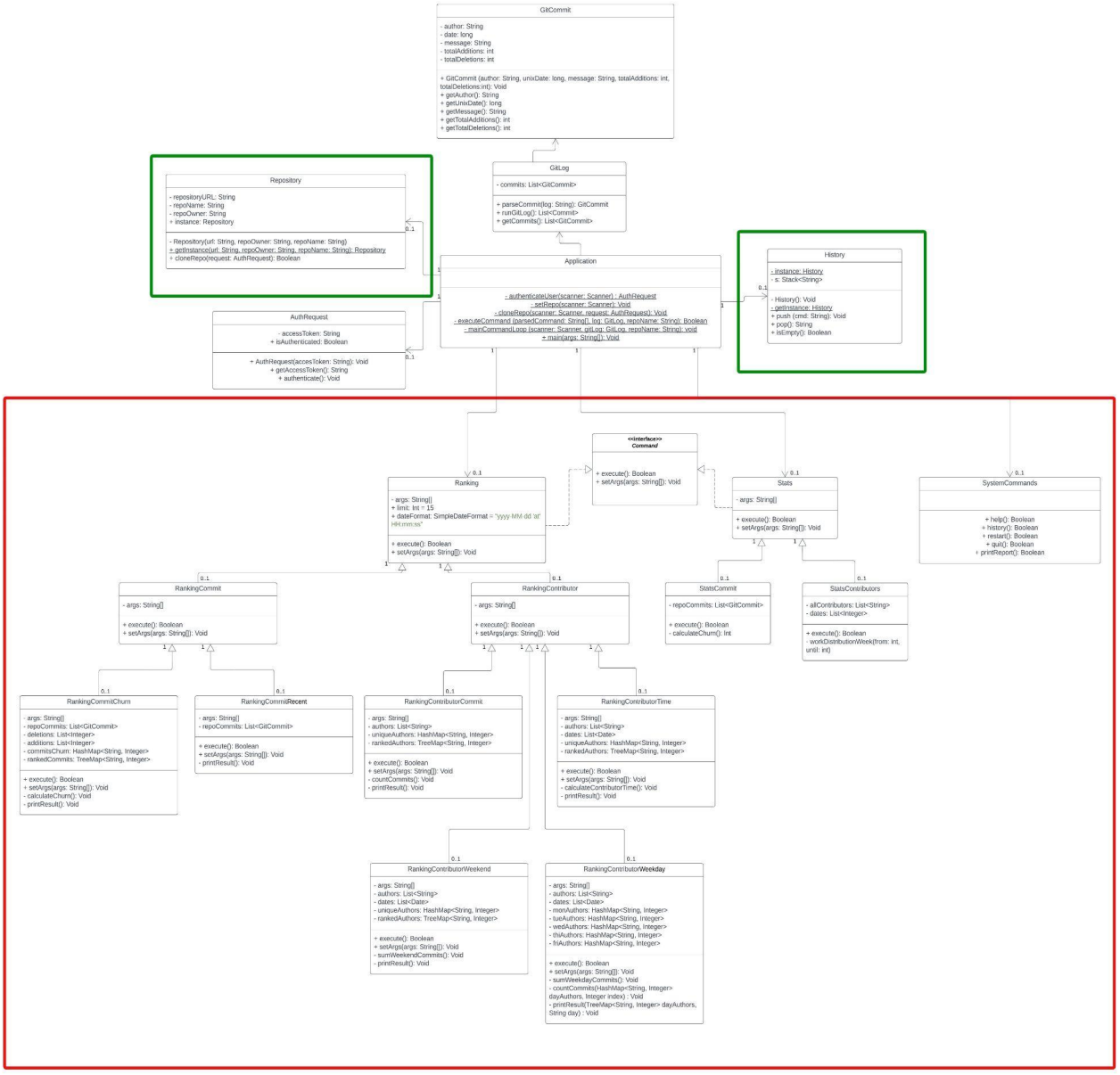
Author(s): all

- Class Diagram
    - Addition of new classes (GitLog, GitCommit) to represent the new approach
    - Addition of new private methods in each class to represent the complex calculations needed in each subcommand
    - Reasoning for why the class diagram is the way it is.
- Object Diagram
    - Highlighted justifications for chosen diagrams
    - Added titles to the diagrams
    - Demonstrated a different chain of commands for diagram 2
    - Corrected typos
- State machine diagrams
    - Added reasoning after design choices
    - Revised the Initializing > Authentication > Cloning transitions in the Application class diagram to match the implementation
    - Minor changes with variable naming to match the implementation
    - Revised + small changes in the Repository and AuthRequest classes diagrams
- Sequence diagrams
    - Change alt fragments to break fragments in order to exit the loops
    - Added explanation for the different colours that were used for actors
    - Made input generic before entering loop
    - Added either a note or an explicit return arrow to model the return of void functions

Revised class diagram

Author(s): Zohaib Zaheer, Vincent Kohm

Link to diagram: [Class Diagram]



The above diagram contains a revised version of our class diagram. In the following section we will underline the most significant changes and discuss why the changes were made. To start with, our Repository class presented us with the opportunity to implement the Singleton Design Pattern. This design choice is explained later in the report. However, this means that the Repository class now has an attribute called instance and also a getter method following the Singleton design pattern. We made changes to the name of the AuthUser class and converted it to AuthRequest as we believed this name is more descriptive of the class. Our

most significant addition is the GitLog and GitCommit class. This changes the approach from running individual *git log* at each execute and instead running logging all commits at once on startup. This is because running git log at each execute requires a lot of compute overhead as we would need to parse logs at each execute call. The original decision to run *git log* was based on using pipes with *git log* but we realized that this would not work as using pipes from within Java required using multiple threads which would require us to debug concurrency issues. That would be very difficult therefore we chose a different route.

The GitLog class is responsible to run the git log command in the process builder and then parse the information obtained into our GitCommit class. We decided to define all attributes of the GitCommit class to be private to follow an immutable approach and also make the program safer from unwanted changes. A series of getter methods was implemented to gain access to all attributes.

Something worth noting is that Application is no longer being generalised by the other classes in its immediate vicinity. This is because we realized that we did not want to create any global variables that could be accessed by any class that is connected to Application. This is because all classes connect to Application, it is the main class with which all user interaction happens and we wanted to preserve immutaibility – a theme that can be seen across the class diagram and implementation. Therefore we chose to use workarounds to this approach. One of which was parameter passing between functions – GitLog gets passed down to execute. Another was using the singleton pattern – Repository and History are both singleton classes.

Highlighted in green you can see our classes that follow the Singleton design pattern. One of them is our History class. A change here is that we now use a stack for storing and printing history as this is a better representation of what we need – namely, a list of commands in the order in which they were entered. Another notable update is the return value of your methods in the SystemCommands class. Here we decided to return a boolean value in order to ensure that each method executed as expected and that no errors occurred. The boolean return value simplified this approach as the Application class could check that the command executed otherwise it informs the user that the command failed.

Highlighted in red you can see the core of our program: our commands following the Command design pattern. The choice of this design pattern is explained in a later section. We decided to add additional private functions and parameters to the specific commands classes. The main reason for this was to increase the readability of the code by hiding complex calculations and decluttering the execute() function. We opted to do so, instead of including them in the Command interface or in any other “higher” classes that would allow sharing, because not every command ever implemented will use them. We decided that creating excessive functions that “might be useful someday” is not a good approach. This way all the command implementations are contained in their specific class making adding new commands easy and not complicated by not having to browse through countless files to check if a function needed has already been declared and implemented somewhere else. The “higher” classes only take care of parsing and executing the correct commands.

In general, there were not massive changes made to our class diagram. This is because we had already implemented the command design pattern in the last assignment. That design

choice allowed us to more easily implement our program very easily without too many unforeseen consequences. One of the biggest advantages of Command was the separation of concerns as this allowed each developer to work on individual commands/individual classes without disrupting the other's work.

Application of design patterns

Author(s): Naomi, Zohaib

	DP1
Design pattern	Command
Problem	All the commands have their own complex executions and may be called by multiple systems across the project. Firstly, how can we make them both accessible from everywhere yet also abstract their inner workings from the invoker? Secondly, how can we ensure the program is extensible and any new command can be added at any time?
Solution	The command design pattern's use of a Command interface allowed us to create different commands with the same functionality, abstracting away the need for the client to know what is going on inside each individual command. The interface also makes sure that all future commands implement the same blueprint which will allow the maintainers of Application to simply add new commands without knowing how they works. Moreover, this also allowed for separation of concerns as we could be sure that each command would have a same execution pattern therefore the developer of the *report* system command did not need to worry about how *ranking* was implemented and could just create an instance of *ranking* and call *execute()* as needed. Finally, use of command design pattern also means that the system is robust enough that any future command could as easily be implemented without bothering the business logic used by GitLog, GitCommit, Repository or any of the Command classes.
Intended use	When the user (client) invokes a command, it is ingested and parsed by Application. Application parses the creates a new instance of Command based on the invoked command and calls execute on that instance.
Constraints	1. Trying to pass variables needed by each command's execute function caused too many parameters to be passed per execute. We fixed this by using the singleton design pattern. This could have also been fixed by using global variables in Application and using them in each of the subclasses but we chose to avoid global variables. 2. System commands could not be of type Command as their execution is different from the normal git commands (for example, they do not have any arguments). Therefore our History class uses a stack of type String and not a stack of type Command.

The second design pattern that we decided to implement, is the creational DP Singleton. We made Singletons out of two of our classes, namely *Repository* and *History*. Consequently, the following table includes both applications of this pattern.

	DP2	
Design pattern	Singleton	
Class	Repository	History
Problem	It is possible to create multiple instances of this class, even though every object would describe the same repository that was cloned from GitHub to the user's machine. In addition, the Repository object needs to be accessed in multiple different points in the program, for instance in the Application class, as well as the Stats class. The fact that various different methods in Application and SystemCommands need information stored in Repository, adds complexity through parameter passing.	Always creating a new object would defy the purpose of the command history functionality. There is going to be only one history at a time. Passing that object to the respective methods leads to an unnecessarily big amount of arguments in chained calls, when only the last method actually needs the History. To illustrate, pushing a command to the History stack, needs to be done at the most specific moment. For example, to push 'ranking contributor commit' to the history, the object would need to be passed from the main loop in Application, to the executeCommand() in Application, to the execute() in Ranking, then RankingContributor and then finally the full command could be pushed.
Solution	Using the Singleton DP solves these issues since it ensures that… 1) there is only one instance of the respective class at a time. 2) the same object can be retrieved from different points in the program. I.e. the Singleton serves as a global instance that can easily be accessed from different classes and methods. 3) the singleton object is only initialised once, when it is requested.	
Intended use	The Repository class stores information about the actual repository, such as its name and owner. In addition, this class is responsible for the cloning process. The Repository object needs to be accessible from different methods during run-time in Application, for example, to initialise the variables. The system can simply access the Singleton object whenever its attributes or functions are needed.	Instead of passing the object from execute method to execute() method, one can simply get the object with History.getInstance() whenever it is needed. At run-time when specific commands are executed, the Singleton is accessed to push the commands to the history stack. This can be seen in the interaction modelled in the "Stats" sequence diagram.

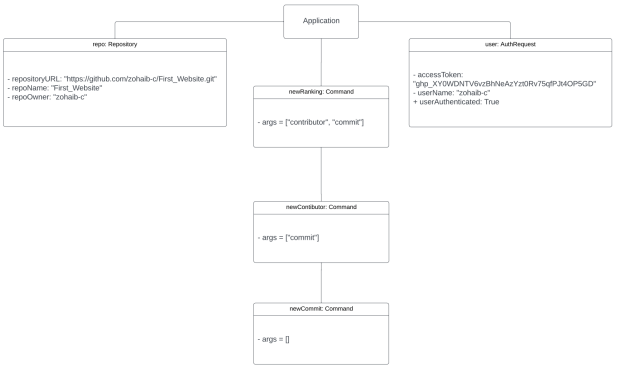
Constraints	There are no additional constraints that the application of the Singleton design pattern is imposing.	If the GitHub Miner were to be implemented with multiple threads, allowing for multiple cloning requests at once, the implementation of the History Singleton would need to be changed. Otherwise, there would be a command history for each thread.

Revised Object diagram

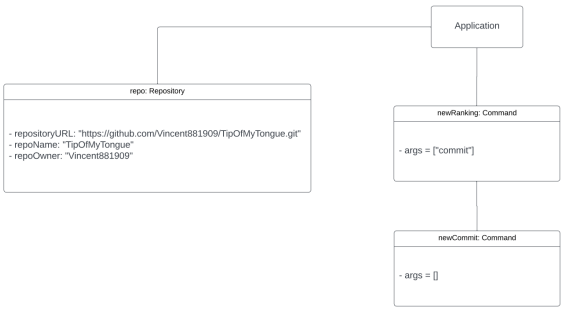
Author(s): Vincent Kohm

[Link to Object Diagram 1]
[Link to Object Diagram 2]

User Authenticated



User not Authenticated



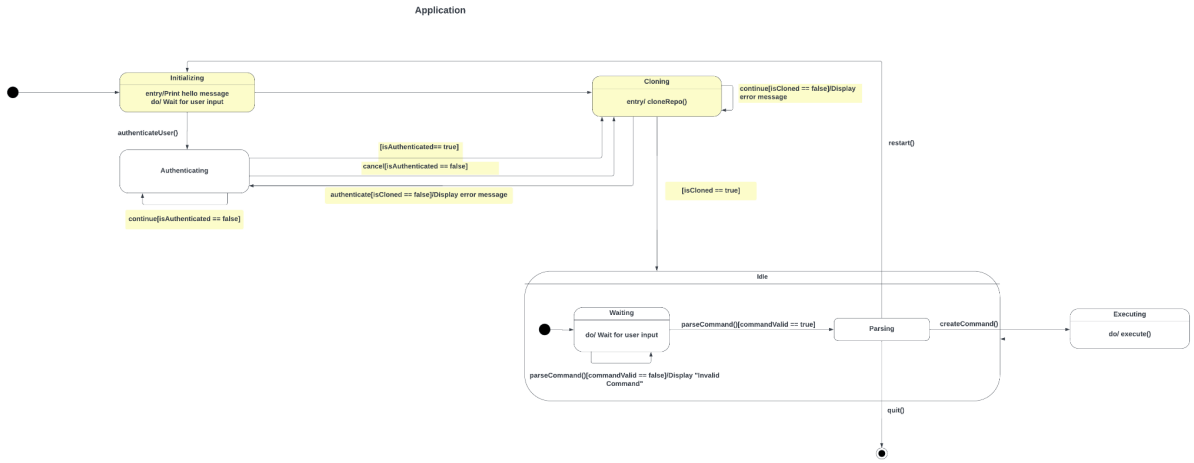
Description

Both of the presented object diagrams illustrate the Application class executing a command, providing insight into different levels of command chains and authentication status. In the **User Authenticated** diagram, the chosen command is "ranking contributor commit," which involves a lengthy chain of commands. This sequence involves the creation of a new instance of Ranking, which then creates an instance of RankingContributor, followed by the creation of a new instance of RankingContributorCommit. Once all arguments are exhausted, the entire chain of commands is executed. This diagram aims to demonstrate the bonus feature of authenticated user status. One of the primary reasons for presenting the **User Authenticated** diagram is to emphasise that our authentication implementation allows users to clone private repositories, which can be particularly beneficial for users working with private repositories. Therefore, it is imperative to showcase this feature in one of our object diagrams. Note that the Repository class includes an attribute named "instance," following a Singleton design pattern, which is omitted from the diagrams as it is deemed non-essential to the viewer. The **User Not Authenticated** diagram highlights a different command chain, with only one

sub-command in its arguments. The purpose of selecting this command in the diagram is to exhibit a different level of the command tree in comparison to the previous diagram. Additionally, this diagram depicts a scenario where the user is not authenticated, which is why no instance of AuthRequest is created. The repository that has been cloned is public and named "TipOfMyTongue." Both diagrams portray the Repository class, which holds the repository's url, name, and owner attributes.

Revised state machine diagrams

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



At the very beginning, it is worth mentioning that we decided to leave some abstractions in the state machine diagram (ex. Event calls) that do not have the same named function reflected in the code. It is explained how those were implemented instead. This was a conscious decision based on the fact that those abstractions make the diagram easier to read and understand rather than trying to represent it exactly with the same functions as in the code. The changes made are highlighted in yellow.

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,** *authenticateUser()*) or skip this step and provide a URL for a public repository (state transition to **Cloning**). This is implemented by having the user decide by input (“yes/no”) whether they want to authenticate or not. The reason we decided to make this the initial state, instead of **Cloning** or **Authenticating**, was to make a clear distinction between the state before cloning the repository and after (**Idle**). This design also allows us to use this state as the “Application set up” state useful for the *restart()* function.

If the user chooses to authenticate, the *authenticateUser()* function is called triggering state transition to **Authenticating**. We decided to not make it possible to go back to **Initializing** as it became (as mentioned before) the “set up” state that should be only accessed once when the application is booting (or after restart)**.** Instead, the user is now placed into a loop

between **Authenticating** and **Cloning.** The user can go back and forth between failed Authentication and Cloning indefinitely or (preferably) until they succeed **-** reasoning after this is explained below**.** If the authentication fails, guard [*isAuthenticated == false]*, the user can either choose to *continue* and stay in **Authenticating** to try again or *cancel* to move on to **Cloning** instead. In the implementation those are not specific functions, but a “yes/no” input from the user. If the authentication succeeds and the guard [*isAuthenticated == true]* a state transition to **Cloning** occurs.

While in the “Cloning” state *cloneRepo()* is executed and the user has to provide the repo’s URL. If the cloning fails, *[isCloned == false]* an error message is displayed and there are two things that can happen. The user will be prompted if they want to try authentication again (yes/no input). If they choose to do so a transition back to the **Authenticating** state occurs. This is important considering the fact that the cloning could fail, because the repository is private. If that happens, going back to the **Authenticating** state allows the user to authenticate before they attempt cloning again. In this case the same rules as above apply - the user can attempt authentication until they succeed or decide to cancel. After that the user can provide a different, correct, URL. If they choose not to authenticate they can proceed with providing another URL. Otherwise, if the cloning was successful and *[isCloned == true]* then a transition to **Idle** occurs. We decided not to print the list of the commands on transition, but instead print a statement "To see a list of commands, please enter 'help'.", forcing the user to try out the command interface to see how it works.

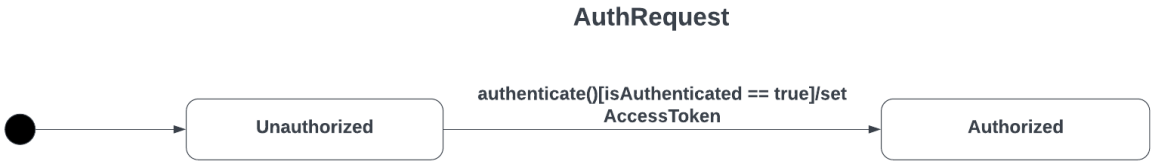
The **Idle** state (represented by the “mainCommandLoop" in the code) is a composite state with two substates - **Waiting** and **Parsing**. The main point of this state is to distinguish the time where Application waits for the user input and the time where it executes commands. We decided to put both **Waiting** and **Parsing** into this composite state, instead of a sequence of separate states, because until command is determined as valid (i.e parsed) the execution won’t happen, which means it would technically still be (conceptually) **Idle**.

After [*isCloned == 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 command gets parsed (also, not a particular function in the code, but this abstraction is suitable for the diagram). If the guard [*commandValid]* is true a transition to **Parsing** occurs. If the guard [*commandValid]* evaluates to false the state stays as **Waiting** and an error message is displayed. 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**. What is worth mentioning is that the[*commandValid]* guards from the diagram were instead replaced by a switch statement in the implementation. This allowed us to reduce overhead and unnecessary conditions in the code. We decided to leave the diagram as is, because the reasoning stays the same and it is a good representation. If the command is not valid it will get rejected (default case), followed by an error message.

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. Restart or quitting makes exiting the composite state faster as it is not needed to go to the **Executing** state. For any other commands, *createCommand() (*also, not a

particular function in the implementation, but rather creating a *new* instance of a Command/System Command) 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**. The execute functions are tied to the commands rather than being a separate function in the Application class.

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 *setRepo()* evaluates to true. This function replaced the initial ValidURL guard. If it evaluates to false the transition does not occur. On entry to the **Cloned** state cloneRepo() is called and the repository gets cloned. Transition from the **Cloned** to the **Empty** state is now not possible. This is because, in an event where it is necessary to clone a new repository the user has to either quit or restart the program. In the latter case a new instance of the entire program is created and thus a new instance of the repository is created with the default value Empty. When the user quits the application all objects are discarded.



The AuthRequest 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 *isAuthenticated* is true - which only happens when authentication succeeds. A change was made and the AccessToken is now set on transition instead of on entry to **Authorized**. This is because the AccessToken is actually used to check whether the authentication was correct. If the authentication fails the transition does not occur AuthUser waits for another attempt (*authenticate()* call).

Revised sequence diagrams

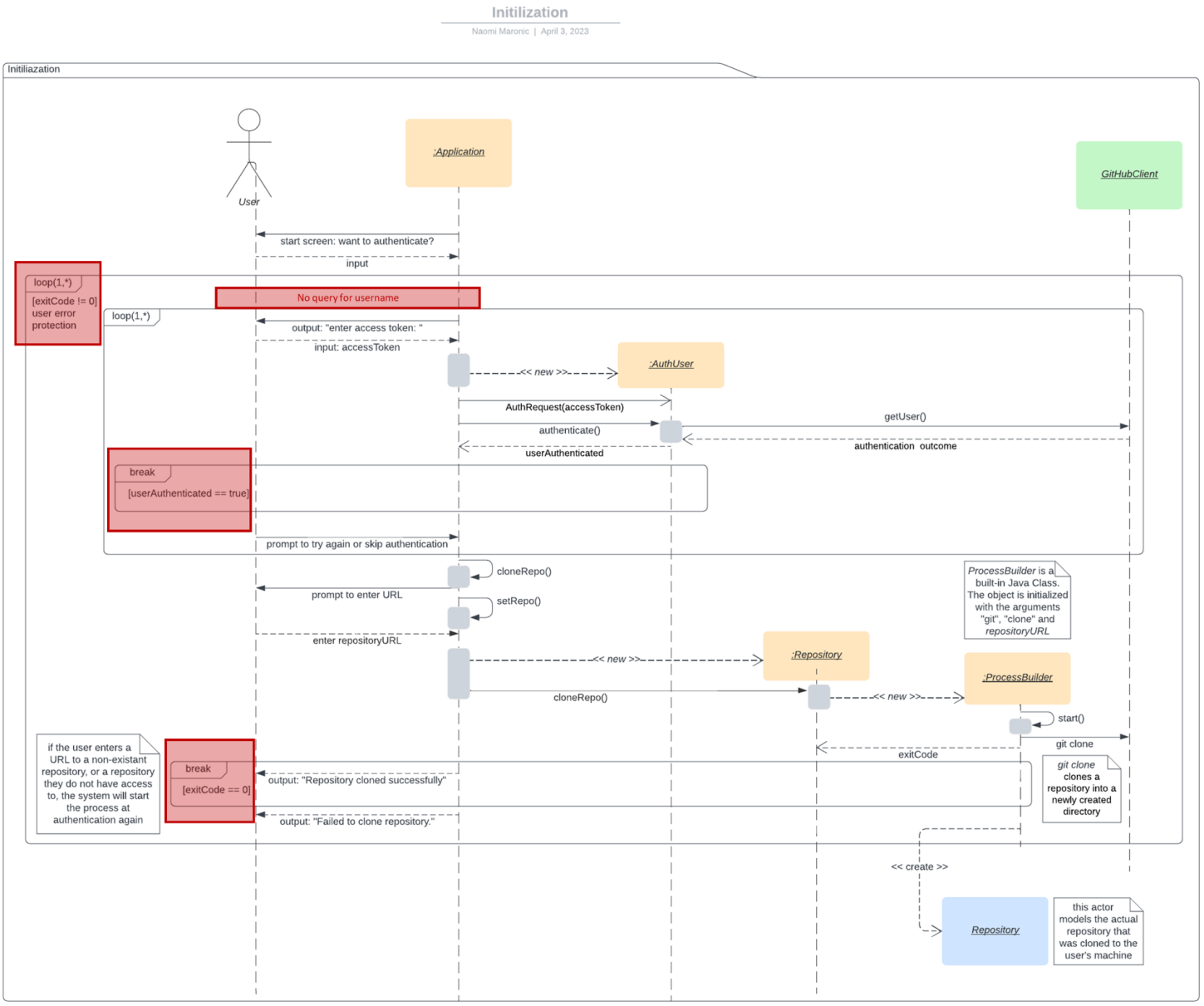
Author: Naomi

We decided to use the sequence diagrams to model the initialization of the system, the parsing of the information returned by *git log* 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 sent 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

Available [here]

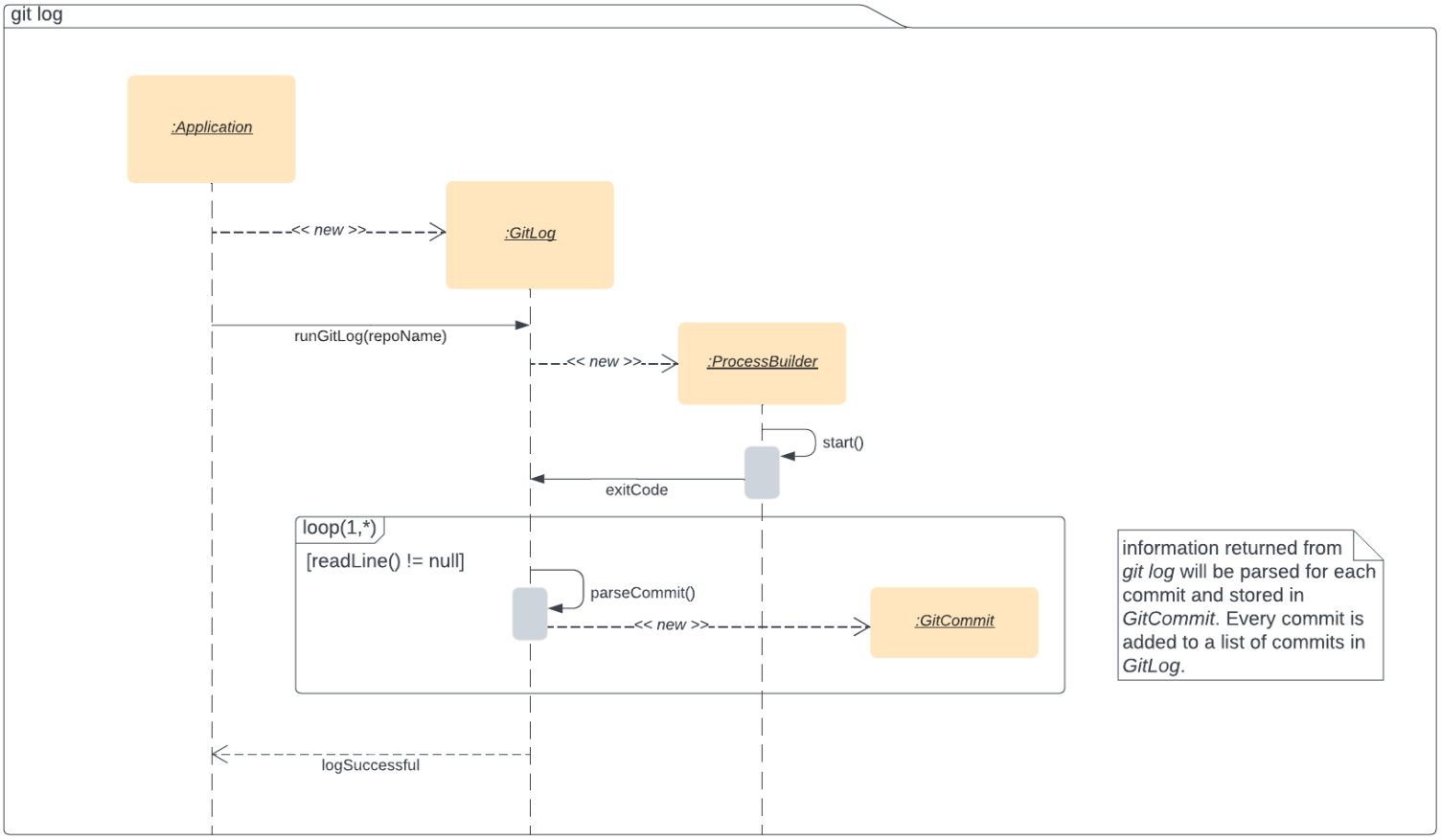


Before describing the modelled interaction, it is important to mention the meaning of different coloured actors. Generally, we decided to model different types of actors in different colours to highlight their purpose in the diagram. Orange actors specify objects of classes that we specified in the GitHub Miner Code. The blue Repository models the actual repository that has been cloned to the user's machine, in contrast to the *Repository* object that is storing information, like the name and owner, of the repository. The different colours are supposed to make their distinction more clear. Lastly, the *GitHub Client* is an external entity that our software is accessing and as such is also modelled in a different colour.
Furthermore, the changes made after Assignment 2 are highlighted with the red boxes.

We assume that the user has started their GitHub Miner Program. The modelled interaction begins with the *Application* object (more specifically in the *authenticateUser()* method) 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.
In the following, a user-error-protection loop is entered that handles the authentication 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. A change from Assignment 2 is that now the entire initialization process including the authentication is included in the loop. We chose to do that since the cloning might have failed because the user forgot to authenticate to access one of their private repositories. It is important to mention that while this is modelled as a loop in the sequence diagram, in the code, after failed cloning the system returns to *authenticateUser()* with a function call. Once the cloning of the specified repository was successful, the loop is exited.
Going back to the beginning of the loop, 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 access token. We omitted the prompt to ask for the user's GitHub username, since it turned out to be redundant. Afterwards, a new *AuthUser* object is invoked. Here, we deleted the *createAuthRequest* function since this is already done in *authenticateUser()*. The *Application* object then initialises the *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*.
If the authentication failed, the user can input whether they want to try again or skip the authentication.
After a successful 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.
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. Instead of using an alt statement to model this, we used a break statement. 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.
To come to a conclusion, the initialization consists of an optional authentication process and the cloning of the repository in question.

Git log

Available here



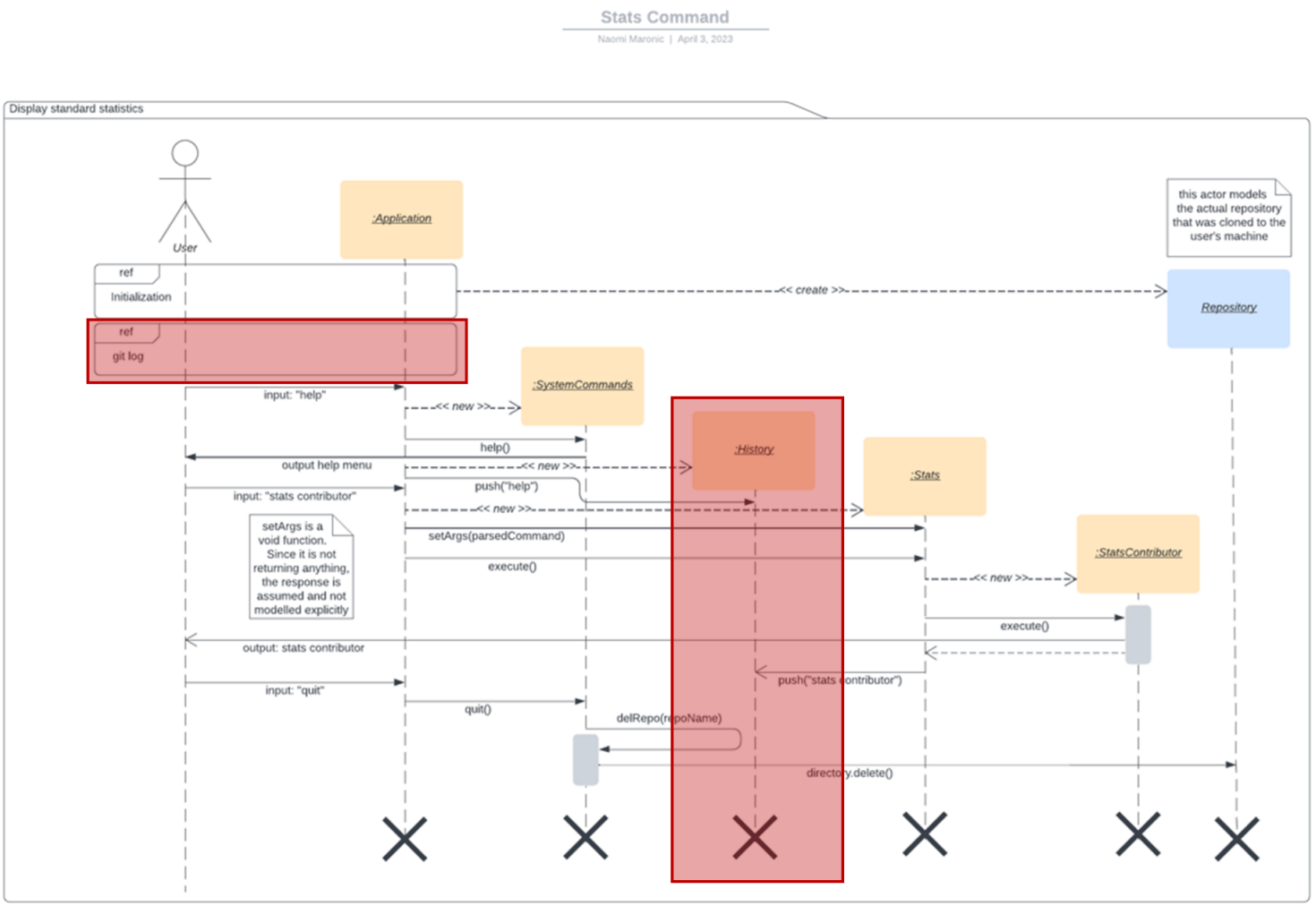
This interaction shows the execution of the *git log* command as well as the parsing of the returned information into the *GitCommit* class. We decided to model this process in a sequence diagram, since it is a vital part of our program. During every mining request, git log is executed and the retrieved information is parsed into a list of commits.
The modelled interaction begins with the *Application* object creating an instance of *GitLog*. It then calls the *runGitLog()* method which leads to another use of the a *ProcessBuilder* object performing the actual *git log*. Afterwards, the retrieved information is processed line-by-line and then stored in instances of *GitCommit.* Each instance is in a complete list of commits for that repository.

During the design process, we took multiple approaches for *git log* into consideration. One thought was to have each command perform a very specific *git log* with a lot of arguments. The advantage here is that there is not a lot of parsing needed. However, this would lead to a lot of calculations during the run time and potentially even the repetition of these computations, since we did not plan to store the results in variables. This approach, performing a general git log in the beginning and storing the information about the commits in a list of objects, circumvents these issues. It also follows the single responsibility principle, since each class is only responsible for one thing, instead of having the command classes performing *git log* as well.

Display Standard Statistics

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

Available here



Note: The previously described colour scheme applies here as well.

After the initialization process and the parsing of the from *git log* returned information (which was modelled in the first two sequence diagrams) the user can input commands to receive information about the cloned repository. The entire interaction takes place in the *mainCommand* loop that is calling *executeCommand* after every user input. The latter method then handles the input and instantiates an object of the respective class.

In this internaction 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. After execution, the help command is also pushed on to the *History* stack. Afterwards, the user enters the “stats contributors” command. This leads to the creation of the *Stats* object. Still in the *Application* object, the argument string is passed to *Stats*. The *setArgs* function, initialises the variables in the newly created object of the subclass, here *Stats*. Directly after, the *execute* member function of the *Stats* object is called where the next part of the command (here: “contributors) is parsed which then leads to the instantiation of the *StatsContributors* object. The inner workings of the *execute* function are omitted in order to reduce the complexity of the UML diagram. 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. 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. This results in the cloned repository being deleted from the user’s machine with a call to *directory.delete()*.When the GitHub Miner application stops running, all objects get destroyed since no data is permanently stored.

Implementation

Author(s): all

We started with the implementation by creating all the classes we modelled in our class diagram. Afterwards, we created the respective attributes and boilerplate methods, to make sure we are not forgetting about anything that was previously modelled. Afterwards, we followed the first sequence diagram that is showing the starting interaction of the user with the system. In other words, we began to implement the *Application* class followed by the *AuthRequest* and *GitLog* classes.

Application

Application is our implementation’s main execution class. One of the main design decisions we made at the very start was that all user direct interaction – which in our case involves only input from user – would happen at Application. This ensures extensibility as Application will need minimal refactoring to add additional functionality. This also ensures that too many classes are not doing the same thing which would cause issues with maintainability as the program becomes difficult to debug – if we allowed multiple classes to take input it would be impossible to tell where the user interaction error is.

It is worth noting that the one place where we waiver from this principle is executing of commands. We felt that it was overly complex to return printable strings (or whatever the result of each individual *execute* is) to Application and we decided it was better to handle the whole execution, which included printing the command response, directly from the *execute* of each command.

Application contains the *main* function which is what gets executed. We used dependency injection to use a single *Scanner* object that reads input from the user by using the scanner object as a parameter to all the other functions that take input in Application. This was done as closing a *Scanner* can only be done once, trying to reopen it causes stream errors.

The *mainCommandLoop* function is our main infinite input loop. It continuously asks for commands from the user, parses the command, and executes it. For execution, we use a switch statement that matches each input to its corresponding command and calls its corresponding command's execute function. This implementation, once again, allows for the developer of Application to not know how exactly the execute function works. They just need to know that they must pass the log to it but they do not even need to know what the log is. Other functions in Application perform validation on user input, call *cloneRepo* of *Repository*, call *runGitLog*, and call the *authenticate* function of *AuthRequest* before moving on to the *mainCommandLoop*.

Cloning

The Repository class is used to store the cloned repository. We use ProcessBuilder to run the git clone command, using two different commands depending on whether or not the user has authenticated themselves.

Logging

Our first challenge was figuring out an apt *git log* command that would give us an output that would be easy to parse into a GitCommit object. We could not just simply use git log and parse the output, as not only did our commands did not need all the results the usual git log outputs, the normal log output is also very difficult to parse by a machine.

Therefore, we landed on the following command to get all the attributes we needed:

```
"git log --pretty=format:--BEGIN--\n%an\t%at\t%s --numstat"
```

Breakdown of the git log command:
The *--BEGIN--* allows us to know when each new commit has begun, and helps with parsing. The \t refers to tabs and adds a tab between each of the returned attributes which also helps with parsing (see below).

This command uses the --pretty=format argument with placeholders to get the following information:

- *an* = commit author name
- *at* = unix time of commit
- *s* = commit message subject

Another challenge was that our commands needed the total number of additions and deletions in each commit but there was no placeholder for those attributes in the pretty

format argument. Therefore, we had to use the *--numstat* argument to git log. This argument returns stats in the following format:

```
210     0       htmlCSS.css
147     0       index.html
153     0       script.js
```

The first column shows the additions per file, the second column shows the deletions per file, and the third column shows the corresponding file.

In the end, a log looks something like this:

```
--BEGIN-
Zohaib Zaheer    1650729346    Add files via upload
210     0       htmlCSS.css
147     0       index.html
153     0       script.js

--BEGIN-
Zohaib Zaheer    1650729259    Initial commit
2       0       README.md
```

Parsing
With bigger repositories came the challenge of 100s of commits and parsing them all into our GitCommit objects. To accomplish this, we use two nested while loops. The outer loop reads until the input is null. The inner loop reads each commit, effectively reading until the next --BEGIN-- can be found. We then use a StringBuilder to create a single line commit with the file information and the commit information. We parse it by splitting the string by “\t” and storing the contents in an array. The contents of this array are funneled directly into GitCommit objects.

Authenticating

We use a library, Eclipse git, to help us authenticate the user. The library abstracts most of the authenticating process. It sends the HTTP request to GitHub to authenticate the user, and if it is successful, it returns the user’s details to us. Otherwise, the user cannot be found and the authentication fails.

Commands

Afterwards, we worked on the individual commands. During the implementation we stumbled into a few unforeseen issues. For example, we noticed that the instance of the *Repository* object is accessed in multiple different points of the program. Since we only have one object representing the locally cloned repository, this object would have needed to be passed from method to method until it reaches the place that requires access to it. After revising design patterns that offer solutions to common issues, we realised that the Singleton DP is the perfect way to solve that problem. In addition, it also ensures that there is always only one object of the *Repository* class.

All the commands used for extracting information from the repo (Ranking and Stats) are implementing the Command interface. The structure of those commands is meant to work in a way that if a user does not specify any arguments all the possible commands "under" the one specified are executed. Ex. if user types in Ranking Commit both Ranking Commit Churn and Ranking Commit Recent will be executed. In the implementation this has been achieved by parsing the command arguments and on each “level” checking if there are no left. If additional arguments are provided a switch statement checks which command should be executed - if the argument provided by the user is not a valid command an error message will be displayed (default case).

The Command interface only includes the setArgs() and execute() function. All of the other functions needed to execute the commands are contained in their designated class (ex. all of the functions needed for Ranking Commit Recent are private functions contained in the RankingCommitRecent class). We are aware that some of the commands share the methods needed for their implementation, ex. both Stats Commits and Ranking Commit Churn share similar calculations. As explained in the Class Diagram section, we opted to not include them in the Command interface or in any other “higher” classes that would allow sharing, because not every command ever implemented will use them. We decided that creating excessive functions that “might be useful someday” is not a good approach. This way all the command implementations are contained in their specific class making adding new commands easy and not complicated by not having to browse through countless files to check if a function needed has already been declared and implemented somewhere else. The “higher” classes only take care of parsing and executing the correct commands.

The implementation for our SystemCommands class consists of five essential methods and some helper functions. The restart method allows the user to restart the program and caused some difficulties when implementing because it turned out to be more difficult than imagined to restart the program and create a new instance of it. The approach we took here was to locate the .jar file and use the process builder to execute the program again. However, with this approach a new application opened which was not desired. Therefore, we had to inherit input, output, and error stream from the parent process and inherit them to the created child process. With this implementation the user can use the same window was before and can safely restart the program. Our help method simply prints a small guide to the user to assist in our program. Our aim here was to be fairly detailed to help the user understand our program and its capabilities better. The implementation of quit challenged us to delete the cloned repository. This process required two helper functions namely deleteRepo() and helperDelRepo(). The former checks if the repository even exists and then calls the latter function to recursively delete all files in the repository similar to the linus command “rm -r”. Lastly the implementation of our bonus feature faced us with the question on whether to add additional reportPrintFunctions to each command. After a few discussions we had the idea to redirect our output stream and then literally just call out command classes and their respective execute functions. This way, the program did not need any major changes and the output was simply re-directed to a file called report.txt. When calling the report function the user can input a series of commands and proceed to create a report by typing quit to exit the list of commands. Like mentioned before, the program redirects the output stream before executing the listed commands. The program then opens the generated report for the user to inspect and save and then the output stream gets restored to its original reference. A few challenges we faced here included changing the output

stream back to its original, restoring the contents of the report.txt file if it already existed, and opening the report on the user's local machine.

Demo video:

https://youtu.be/D7WHjiN3_Hw

Time logs

Time Log