

Assignment 2

Team number: 52

Team members

Name	Student Nr.	Email
Joachim Bose	2736851	joachimbose48@gmail.com
Márkó Vlachopoulos	2735992	m.d.vlachopoulos@student.vu.nl
M. Z. Y. Ali (Zain)	2812263	m.z.y.ali@student.vu.nl
Ammar Obeid	2661793	a.obeid@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): Ammar, Joachim, Márkó, Zain

- Refactored statistics into 3x Git, 3x GitHub
- Rephrased features into user story format
- Compacted overview into a high-level description
 - More details for stakeholders
- Fixed duplicate QR IDs
- Fixed QR categories
- Made standalone terms like “easy”, “productive” and “fair” more specific
- Made QR5 (Security) more general
- Addressed unspecific meeting times
- Addressed workload dispute disagreements

Class diagram

Author(s): Zain, Joachim

Link to draw.io diagram [here](#), rendered diagrams at [docs/diagrams/class-diagram](#)

Class: **Application**

Represents the core framework of the application. It ties together the **Repository** class (representing Git data) and the command classes. It handles the initialisation of the **Repository** class (by requesting input from the user) (see also for more details: description of the **Repository**). It also acts as the main “command handler” thread, requesting command input from the user, parsing it, selecting the right command class to execute, and preparing the arguments for this command class based on its argument parser (see **Command** class).

Class: **Repository**

Represents the cloned repository. It contains a path to the repository on disk, as well as the URL of the repository. It also contains a map of branches.

The constructor of this repository handles the cloning of the repository and progress display. The repository has a *switchActiveBranch()* method. This method sets the *activeBranch* field of the repository, and if the branch doesn't exist in the map of branches yet, the branch is created and initialised. If the method is called with the value *null*, the method is called with the name of the default branch of the repository.

The *destroy* method deletes the Git repository on disk, using the private *filePath* field.

The **Repository**, **Branch** and **Commit** classes have been designed in such a way to abstract away the idea of the active branch, or the idea of a branch at all, for the purpose of calculating statistics. Statistics don't care about repositories or branches, they just need commits, and the **Repository**'s *getCommits()* method takes care of selecting the active branch and returning its commits. In other words, all of the **Commit**, **Branch** data structures are hidden complexities from the perspective of other modules like the statistical calculation and UI.

Class: **Branch**

This class represents a single branch of a cloned Git repository. It contains a list of commits, and a method that initialises this list by parsing the output of the git log command.

Class: **Commit**

This class represents a single commit. It contains the relevant information about a Git commit, such as the commit description, author, and the number of added and removed lines.

Class: **TerminalIO**

This class is a utility class with static methods that allows basic I/O with the terminal (reading, writing, and a utility *prompt()* method that performs a write and then a read).

This class hides all the information having to do with terminals in java, the methods are not insanely deep, but the java equivalents are very verbose. This is why we chose to create this class.

Class: **ArgumentParser**

This class allows us to create a parser that can be configured to parse multiple key-value input arguments, with optional validation for specific values, such as a “sort-by” argument that can take “commit” or “loc” values. The class has a *parse()* method, which takes a raw input string, parses the input string to the configured key-value arguments, validates them, and returns the **ParsedArguments** data type.

Datatype: **ParsedArguments**

A *Map<String, String>* used to represent a map of parsed and validated arguments.

The **ArgumentParser** and **ParsedArgument** models have been designed in such a way that commands can define their arguments elegantly. The work of validation or string parsing is nicely abstracted by this class, while also being able to pass around the **ParsedArgument** to child contexts. For example, the **StatisticCommand** needs the *name* argument, but its child context **Statistic** needs the *sort-by* argument; in this case, the **StatisticCommand** can simply pass the **ParsedArgument** to the **Statistic**. It's also noteworthy to state that the **Application** and **ArgumentParser** do not share information about command *structure*; the **ArgumentParser** only knows about arguments, while the **Application** only knows about the top-level command.

Class: **Command**

An abstract class that is implemented by every command in the application. The command has an abstract *name* property, which is the string that the **Application** will use to select this command class, based on the input string provided by the user.

It also has an abstract *argumentParser* field, which command classes can set to an **ArgumentParser** instance in order to configure the arguments this command needs. The **Application** class will later use this *argumentParser* field in order to parse and validate the specific arguments for this class (see **Application**, **ArgumentParser** classes).

The **Command** class also has an abstract *run()* method. The parameter of this method is the **ParsedArguments** data type. This *run()* method is called by the **Application** class as well.

Class: **DeleteRepositoryCommand**

This class extends the abstract **Command** class. The class defines the remove-repo command, which deletes the cloned repository from the user's file system and prompts the user to clone another repository.

Class: **SwitchBranchCommand**

This class extends the abstract **Command** class. The class defines the switch-branch command, which calls the *switchActiveBranch()* method on the Repository.

Class: **StatisticCommand**

This class extends the abstract **Command** class. The class defines the statistic command, which calculates statistics on the active branch of the repository.

The class manages a number of classes that implement the abstract **Statistic** class. Similar to how the **Application** selects a **Command** instance based on its *name* field, the **StatisticCommand** selects a **Statistic** instance based on its *name* field. It then calls the **Statistic** instance's *calculate* method with the **ParsedArguments** passed down to it.

Interface: **Statistic**

This interface represents a statistic that can be calculated on a GitHub repository. It contains a *name* property, which is the name of the statistic (e.g. "most-active-contributors"), which the **StatisticCommand** uses to select this statistic based on the user's input string. The calculation of the statistic happens in the *calculate()* method.

This interface has been modelled in such a way that various kinds of statistics can be calculated, i.e. not restricted to Git or GitHub data. The *calculate()* method returns *void*, which is done to avoid having to pass the calculated statistic result up and down the entire command execution call stack. The *calculate()* method is responsible for formatting and printing of results, however, this does **not** become repetitive because these operations are abstracted with the **Table** and **ProgressBar** utility classes.

Class: **GitHubStatistic**

An abstract class that implements the **Statistic** interface. This class adds the protected *callAPI()* method, which can be called by statistics classes extending this abstract class, to make it easier for them to make requests to the GitHub API.

Class: **GitStatistic**

An abstract class that implements the **Statistic** interface. This class does not contain any functionality, it simply exists to make the separation between GitHub statistics and Git statistics clearer.

Class: **ContributorStatistic**, **BranchStatistic**, **WeekdayStatistic**, **IssueStatistic**, **PullStatistic**, **ReleaseStatistic**

These classes all extend **GitStatistic** or **GitHubStatistic**, and work pretty similarly. In their *calculate()* method, they all get their data from a **Repository**, use the **Table** class to format results, and use the **ProgressBar** class to display progress and results.

Class: **Table**

A utility class that can be used to easily generate ASCII tables.

The class is constructed with a list of table headers.

The class has an *addEntry()* method, which takes a variable amount of arguments, the amount of arguments should be the length of the table header.

Once all of the entries have been added, the *toString()* method can be called to generate an ASCII table.

Class: **ProgressBar**

A utility class for the user interface, that can be used to display a progress bar, and eventually print the results.

This class is constructed with a string argument *taskName()*, which is displayed to the user when the progress bar is started. This starting process happens with the *start()* method.

The method used to increment the progress bar is called *setProgress()*, which takes a single float argument, which is the percentage the bar should be updated to.

Once the calculation of results is done, the *finish()* method should be called, which takes a string input argument. In this method, the progress bar is cleared from the terminal and the given result is printed.

The **Table**, **ProgressBar** and **TerminalIO** user interface utility classes have been designed in such a way that any part of the application that needs to display output isn't tied to the usage of a progress bar or a table. For example, the **Repository** constructor needs to display a progress bar, but no table. Most of the **Statistics** need to display a progress bar and a table. Some other commands need to just print something without a progress bar or a table.

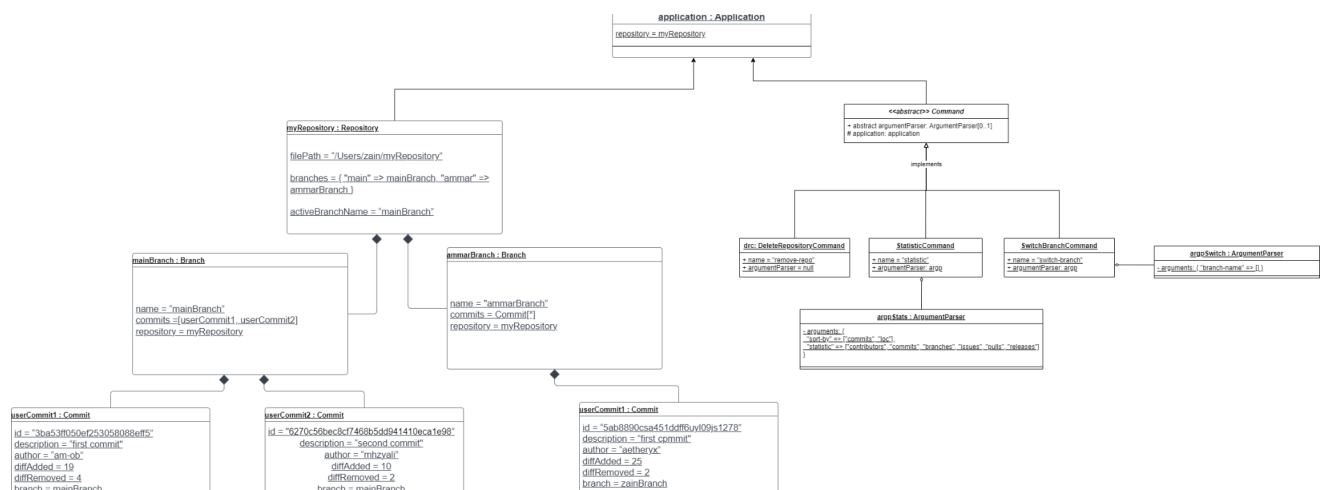
In order to elegantly cover all of these use cases, an output is simply represented as a string, and formatting utilities like **Table** can easily generate their string representations. With this pattern (using simple strings for output), both **TerminalIO** and **ProgressBar** can be implemented abstractly, without needing to care about the format of the output.

Object diagram

Author(s): Ammar Obeid

Link to draw.io diagram [here](#), rendered diagrams at [docs/diagrams/object-diagram](#)

This section illustrates our Object diagram, which describes a snapshot of the system while it is running. As noticed above we have the main class: **Application**. The program will prompt the user for their input. When the user pastes a link of the git repository new class objects will be directly made.



When a certain repository is given to the application a new object of the class **Repository** will be initialised in this case a clone of the original repository and called myRepository. Obviously this repository has two different branches. myRepository has two. mainBranch and ammarBranch, each with its name, commits, and the name of the repository stored in different variables. If one or more commits were done within a branch new objects of the class **Commit** will be made, storing its id, description, author, differences added, things removed, and the name of that specific branch.

We can see that there are three commit objects, two commits within mainBranch and one within ammarBranch.

On the other hand we have an abstract class **Command**. No objects can be created for that class. But an object is created for other related classes.

The UML object diagram above describes the instances of two subsets of the complete class diagram. Design ideas and decisions are already explained in the section of the class diagram.

State machine diagrams

Author(s): Márkó

Link to draw.io diagram [here](#), rendered diagram at [docs/diagrams/state-machine](#)

Repository State Machine

The **Repository** class has four states, each representing a state in which the local repository is in. Not cloned is the default state, the user can only execute statistic related commands if there is a local copy of a repository to work on. When the user provides a valid URL of a Github repository the program creates a local copy of the repository. After successful cloning, input for three types of commands are available.

From the Cloned state the user has the option to remove the repository or exit the program, both of these activities result in the deletion of the repository, after removing a repository the program goes back to the Not Cloned state. The program can also be exited from the Not Cloned state.

Application State Machine

From the Cloned state the application can prompt for a command input. Then the **Application** class decides which **Command** class should be used. There are 3 options: **SwitchBranchCommand**, **RemoveRepositoryCommand** and **StatisticCommand**. The **SwitchBranchCommand** switches to the new branch in Git (specified by the argument) and sets the **Repository** class' *activeBranch* field to this new branch, then goes back to the Cloned state. The **RemoveRepositoryCommand** goes to the Deleting state, and deletes the local copy of the repository. The **Application** selects the **StatisticCommand** class, which then decides which **Statistic** has to be calculated (specified by the argument), calculates the **Statistic**, then goes back to the Cloned state.

Sequence diagrams

Author(s): Joachim with diagramming help from Zain

Sequence diagrams cloning and switchBranch

Link to draw.io diagram [here](#), rendered diagrams at [docs/diagrams/sequence/cloning](#)

As we start the application, the application should ask the user for the url of the github repository they want to investigate, no other commands can be inputted because there is no repo to run any commands on yet. The **Application** then calls the *read method* on the **TerminalIO** class which works like the *input* method from python. **Repository** instance can then be created to represent the cloned repository. The repository then calls *switchActiveBranch()* on itself, instead of the **Application** calling *switchActiveBranch()*, because of the law of demeter. This repository is then cloned while the progress is being updated in the **ProgressBar**. The **ProgressBar** can be started with it's *start()* method, which prints the empty 0% progress bar on the screen. This **ProgressBar** can be updated with *setProgress()* which you can see being used in our loop, to keep the user updated on our progress. We chose this approach, because of the easy state implementation, and easy interface (only 3 methods). Cloning can fail, network errors for example, or invalid urls. In our quality requirements, we need to be able to recover from failures and should therefore have an exception catcher. If the cloning was not successful, we destroy the **Repository** instance and remove the files from disk. After which, we ask the user to input the URL once again and try to clone.

When the clone is successful, the **Repository** class immediately creates a default **Branch**, and populates it. The *processGitLog()* method (again called from the constructor of the branch because of the law of demeter) executes the git log command and parses its output while using the same **ProgressBar** class, as we saw before in the cloning process. Executing the git log command can also fail, if the branch we are trying to log does not exist for example. When git log fails, we delete the entire **Repository** if this is the first branch we clone (which is only the case if we try to clone the default branch and fail). If we were trying to clone anything but the default branch, we switch back to a branch we know exists.

The user can then use the switch-branch command to do the same switching of branches as discussed before, as well as calculating statistics.

Sequence Diagram with Statistic Commands:

Link to draw.io diagram [here](#), rendered diagrams at [docs/diagrams/sequence/commands](#)

The start state for this diagram is a successfully cloned repository, where the user has navigated to the desired branch using switch-branch.

The **Application** calls *promptForCommand()* on itself, which causes it to call *read()* on **TerminalIO**. This then reads the command from the terminal, which is parsed in the **Application** class. The **Application** class uses the parsed command name to select a **Command** class, and uses its *argumentParser* to parse the command's arguments. Then, it can *run()* the **StatisticCommand** with these parsed arguments, which calls *calculate()* on the correct statistic (in this case **MostActiveContributor**). This class then makes a **Table** to fill out the results into and easily stringify their results and starts the **ProgressBar**. Then while calculating, it updates the progressbar to the user with the same interface as we saw in the cloning process and git log parsing process for code duplication reasons. When this fails, it writes the error to the console, when this succeeds it prints the table to the console using the *Table.toString()* method which looks like **TerminalIO.write(Table.toString())** which we found to be very elegant and easy to use.


Deleting repositories in sequence diagram

Link to draw.io diagram [here](#), rendered diagrams at [docs/diagrams/sequence/delete-repo](#)

Then at last, the user deletes his repository to reclaim disk space, after being prompted for another command by the application. The deletion of disk space is first attempted by the **DeleteCommand** class. When this fails (when a file is open in another process for example) the delete returns and another command is prompted after notifying the user why his repo has not yet been deleted. When the deletion is successful, the method returns and the **Repository** instance is destroyed, upon which the **Application** prompts the user for another github url to make our application immediately reusable and to complete the circle.

Time logs

The time logs are on Google Drive here:

 Time log