Assignment 3

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

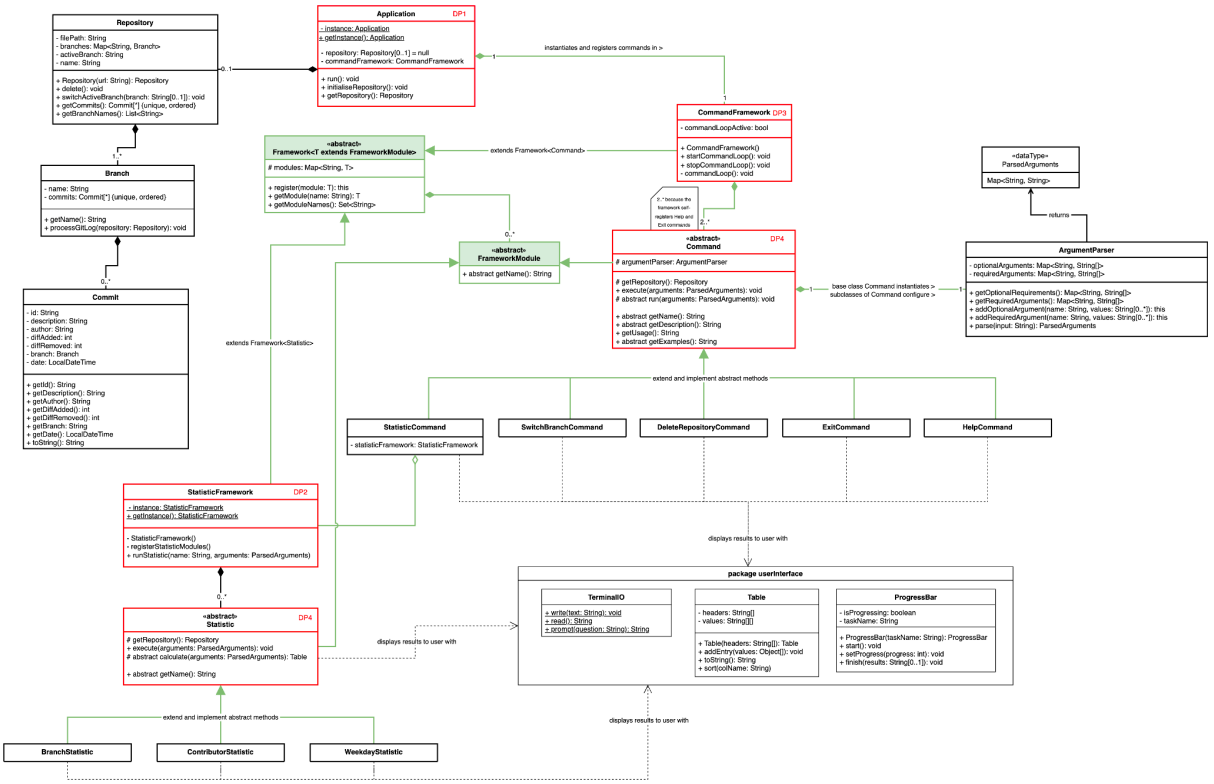
Summary of changes from Assignment 2

Author(s): Joachim, Zain

- Class diagram description
    - Added connections back to the requirements
    - Polished up some design decisions
    - Added **Framework<T>, CommandFramework, StatisticFramework**
    - Refactored **Command** and **Statistic** to be abstract classes, applied the Template Method design pattern to both
- Class diagram
    - Added all private methods we made during the implementation phase.
    - Added useful getters.
    - Added **CommandFramework, Framework<T> and StatisticFramework**
- Object diagram description
    - Complete rewrite of description. To add a load of design decisions
- Object diagram
    - Added some more **Command** subclasses
    - Modelled **StatisticFramework CommandFramework and Framework<T>**
- State machine diagram
    - Untangled the diagrams into two separated diagrams.
    - Updated some names
    - Polished the look
- State machine diagram description
    - Complete rewrite with a load of design changes added.
- Sequence diagram
    - Rewrote the diagrams with more spacing, more activation bars to make them more easy to read.
    - Removed misuse of ref statements.
- Sequence diagram description
    - Complete rewrite adding a load of design decisions.

Revised class diagram

Author(s): Zain, Joachim



Class: **Application**

Represents the core of the application. Ties together the **Repository** class (representing Git data) and the command framework. It handles the initialisation of the **Repository** instance (by requesting input from the user) (see also for more details: description of the **Repository**). Once the Repository instance is initialised, it hands over control to the **CommandFramework** instance, which will start prompting the user for command input.

Class: **Repository**

Represents the cloned repository. Most importantly, it's composed of a number of branches, which are mapped by name. There are methods to manage these branches, such as *getBranchNames()* and *switchActiveBranch()*.
The constructor of the class handles cloning the repository and displaying progress to the user, using the **ProgressBar** class.
The *delete()* method deletes the Git repository on disk, using the private *filePath* field.

The **Repository**, **Branch** and **Commit** together form the module that is responsible for features F1 and F4, cloning and parsing the git repository. These classes have been designed in such a way to abstract away the idea of the active branch, parsing, branches, or interfacing with git at all, for the consumers. The **Repository**'s *getCommits()* method takes care of selecting the active branch and returning its commits. The **Commit, Branch** data structures are hidden complexities from the perspective of other modules.

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 the data of a single Git commit, containing information such as the commit description, author, the number of added and removed lines, etc.

Abstract Class: **Framework<T>**

Represents a generic framework, implementing the Inversion of Control design principle. Stores a number of modules - the interface segregation design principle is applied here by having the modules implement the **FrameworkModule** interface. This interface describes a *getName()* method, which is used to identify modules.

Class: **CommandFramework**

Implements the abstract **Framework** class. The module type is the **Command** class.
In addition to managing **Command** modules, it contains the logic for the main command loop: repeatedly prompting the user for input, selecting the appropriate **Command** module based on this input, executing this module, etc.

This class aims to apply the Inversion of Control design principle to the codebase. By having the **CommandFramework** manage modules but also be in charge of using the modules, the Application does not need to select the command modules in a big if-else chain, and control is inverted.

Class: **Command**

An abstract class that is implemented by commands. The command has an abstract *name* property, which is the string that the **CommandFramework** will use to select this command class, based on the input string provided by the user.
The base class instantiates an *argumentParser* field, which subclasses can optionally configure. The Command's own *execute()* method will later use the *argumentParser* to parse arguments. The Template Method design pattern has been applied here; this *execute()* method calls the protected abstract *run()* template method with the parsed arguments.

Together with the **ArgumentParser** and **ParsedArguments** and the subclasses of **Command,** these classes form the Command handling module responsible for F3: providing the user with helpful familiar command structure to control the application.

Improvements: *argumentParser* is always instantiated and not abstract, subclasses just add arguments

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. Arguments are registered with either the *addOptionalArgument()* or *addRequiredArgument()* methods. The class has a *parse()*

method, which takes a raw input string, parses the input string according 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 with the goal of commands not needing to parse arguments with manual string manipulation. Commands can elegantly configure an ArgumentParser instance, which nicely abstracts the work of validation and string parsing, while also being able to pass around the **ParsedArgument** to child contexts.

Class: **DeleteRepositoryCommand**

Defines the delete command, which deletes the cloned repository from the user's file system and prompts the user to clone another repository.

Class: **SwitchBranchCommand**

Defines the switch-branch command, which calls the *switchActiveBranch()* method on the Repository.
The **SwitchBranchCommand** and **DeleteCommand** class is shallow because it only calls *Delete()* on **Repository,** but this is necessary for the proper hiding of the command implementation and generalisation. Hard coding such commands would be bad for expandability and generalisation, and putting these classes with the **Repository**, would cause information leakage between these two components.

Class: **StatisticCommand**

Defines the statistic command, which calculates statistics on the active branch of the repository and displays them.
The class contains a weak reference to the **StatisticFramework** singleton. Whenever the command logic needs to execute a statistic, it simply calls the *runStatistic()* method of the **StatisticFramework** instance.

Class: **StatisticFramework**

Implements the abstract **Framework** class. The module type is the abstract **Statistic** class. This framework self-registers the **Statistic** modules, because we will never want to manage two separate **StatisticFramework** instances. For that same reason, this class applies the Singleton design pattern.

Abstract class: **Statistic**

This abstract class represents a statistic that can be calculated on a repository. It contains a *name* property, which is the name of the statistic (e.g. "most-active-contributors"), which the **StatisticFramework** uses to select and execute this statistic based on the user's input.

The Template Method design pattern has been applied here; the public *execute()* method calls the protected abstract *calculate()* template method, which returns a table. The super-method sorts and prints this table. The goal is to maintain generalisation within the

modules: the *calculate()* template method simply needs to return a table populated with data, and then the *execute()* method will take care of sorting and printing this data.

Class: **ContributorStatistic**, **BranchStatistic**, **WeekdayStatistic**

These classes all extend **Statistic**, and work pretty similarly. In their *calculate()* method, they all get their data from a **Repository**, and use the **Table** class to store results. The Abstract **Statistic** class then sorts and prints this table.

Class: **TerminalIO**

A static utility class, abstraction layer over terminal I/O (reading, writing, etc).
We chose to create this class because it hides all the information having to do with terminals in Java. The methods are not deep, but the plain Java equivalents are very verbose.

Class: **Table**

A utility class that can be used to easily generate ASCII tables.
Constructed with a list of table headers. Has an *addEntry()* method, which takes a variable amount of arguments, the amount of arguments should be the length of the table header.
The table can be sorted with the *sort()* method, which always sorts the table descendingly by the given column.
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.
Constructed with a string argument *taskName*, which is displayed to the user when the progress bar is started with the *start()* method.
The bar is progressed with *setProgress()*.
Once the operation is done, the *finish()* method should be called, which can optionally take a string argument to print.

The **Table**, **ProgressBar** and **TerminalIO** user interface utility classes have been designed in such a way that systems that need to display output aren't tied to the usage of a progress bar or a table. For example, the **Repository** needs a progress bar, but no table. All **Statistics** need a table, and maybe a progress bar. 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** override *toString()*. 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.

Reasons for changes

Most of these changes are a result of starting our real-world implementation and starting the application of the design patterns. Once we had a Java implementation using our template method design patterns, we saw the "Framework" pattern emerge and implemented this with the abstract **Framework** class.

Application of design patterns

Author(s): Zain, Joachim

	DP1 - Application - Singleton
Design pattern	Singleton
Problem	Multiple classes need to access the Application. They should not each instantiate their own instance - we should only have one instance of the Application class at all times.
Solution	The Singleton design pattern allows us to give out multiple references to a singular instance of the Application.
Intended use	Whenever a class needs to do something with the Application class, it should retrieve the instantiated instance with the Application.getInstance() static method. It should not (can not) instantiate a new Application instance.
Constraints	The constructor of the Application class has been made private, so that classes cannot instantiate a new Application instance.

	DP2 - StatisticFramework - Singleton
Design pattern	Singleton
Problem	The StatisticFramework is a self-registering framework, meaning we will never have two distinct StatisticFrameworks with different modules. We only ever want to have one StatisticFramework with all Statistic modules.
Solution	The Singleton design pattern allows us to ensure that there will only be one instance of the StatisticFramework class.
Intended use	Whenever a class depends on the StatisticFramework, it should retrieve the instantiated instance with the StatisticFramework.getInstance() static method. It should not (can not) instantiate a new StatisticFramework instance.
Constraints	The constructor of the StatisticFramework class has been made private, so that classes cannot instantiate a new StatisticFramework instance.

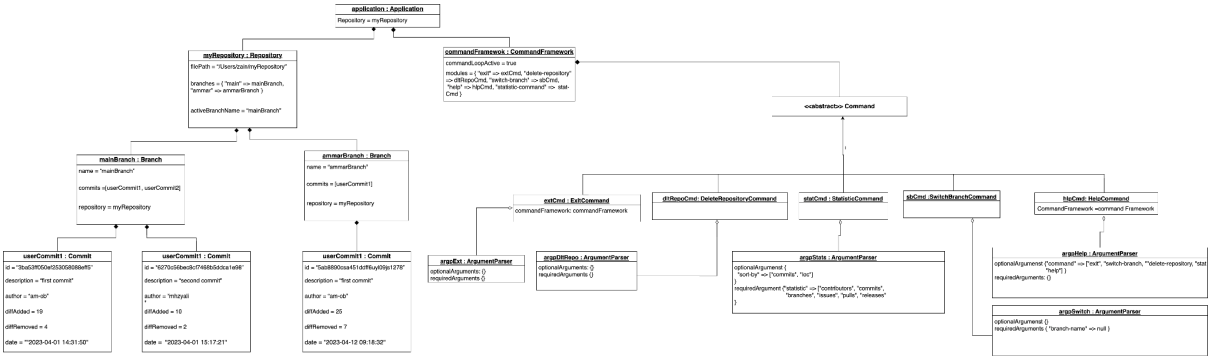
	DP3 - CommandFramework - Inversion of Control
Design principle	Inversion of Control
Problem	Our application contains a number of abstract “command” modules. These commands should be fully independent and unaware of each other. The process of selecting and executing the appropriate command should also be independent of which commands are registered, we want to avoid a giant if-else-chain.
Solution	Applying the Inversion of Control design principle: creating a CommandFramework class, into which the Application can externally register individual command modules. The CommandFramework then dynamically manages these modules, and takes over control from the Application.
Intended use	The Application class should first instantiate its own CommandFramework instance. It can then register its command modules into the framework. Once the modules have been registered, the control can be handed over to the framework by calling the CommandFramework.startCommandLoop() method.
Constraints	Because the CommandFramework instance is stored privately, individual Commands cannot access other commands, i.e. are unaware of their existence. The exception to this rule are the self-registered Help and Exit commands, which can access other commands, by being registered by the framework.
Additional remarks	A design principle rather than a design pattern, but quite significant to the application.

	DP4 - Command - Template Method
Design pattern	Template Method
Problem	The execution of a command always starts with the same set-up logic: parsing the input and handling user facing exceptions.
Solution	The repetitive logic is baked into the public Command.execute() method, and the dynamic per-command logic is moved to the abstract template method Command.run(). The ArgumentParser instance can now also be made protected.
Intended use	The CommandFramework that executes commands should use the Command.execute() method. Subclasses implementing the abstract Command class should implement their logic by overriding the abstract Command.run() template method.
Constraints	The public Command.execute() method is made final, so that the wrong method cannot be overridden.
Additional remarks	The template logic was previously a part of the CommandFramework, meaning that the fields this logic depended on had to be public. In addition to reducing code repetition, applying this design pattern allowed us to reduce the size of the Command's interface, because the logic depending on those fields is now fully internal and can be made protected instead of public.

	DP5 - Statistic - Template Method
Design pattern	Template Method
Problem	The execution of a statistic always ends with the same formatting and printing logic: validating the sort-by argument, sorting the table by the sort-by argument, and then printing the table.
Solution	The repetitive logic is baked into the public Statistic.execute() method, and the dynamic per-statistic logic is moved to the abstract template method Statistic.calculate().
Intended use	The StatisticFramework that executes statistics should use the Statistic.execute() method. Subclasses implementing the abstract Statistic class should implement their logic by overriding the abstract Statistic.calculate() template method.
Constraints	The public Statistic.execute() method is made final, so that the wrong method cannot be overridden.

Revised object diagram

Author(s): Ammar



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 the user inputs a github URL into the system to clone, a new **Repository** instance is created with this url which starts cloning the repository files from the web which it remembers with the file path member. It needs to know the file path because the git checkout, git log, etc have to be run inside the root of the repository. After it is cloned, it immediately starts parsing the default branch (in this case main) and populates its active branch member. It needs to keep track of the active branch because the user wants to be reminded in what branch he is now doing things, besides *switchActiveBranch()* should not do anything if it's already on the same branch to conserve time.

When the user switches between branches, they want to have the least loading time as possible, and therefore the branches should be saved, preferably inside the **Repository** class, because of the role of repository manager it has been given and the information leakage it would imply if the branches were stored anywhere else.
Instances of **Branch** will hold the commits on that branch to store git log data, because having to call git log everytime we want to calculate anything, will be a waste of resources if we can just store it somewhere. Preferably a class on the Branch level of the repository module, because git log is branch specific, and has everything to do with the Repository module and not much to do with other modules. This is why the Branch class is in this spot. We can see that there are three commit objects, two commits within mainBranch and one within ammarBranch. The commits hold all data relevant for performing the statistical features 1, 2, and 3. They hold the branchName as a String instead of a Class, because that would expose the **Branch** class to the scary world of other modules, creating information leakage.

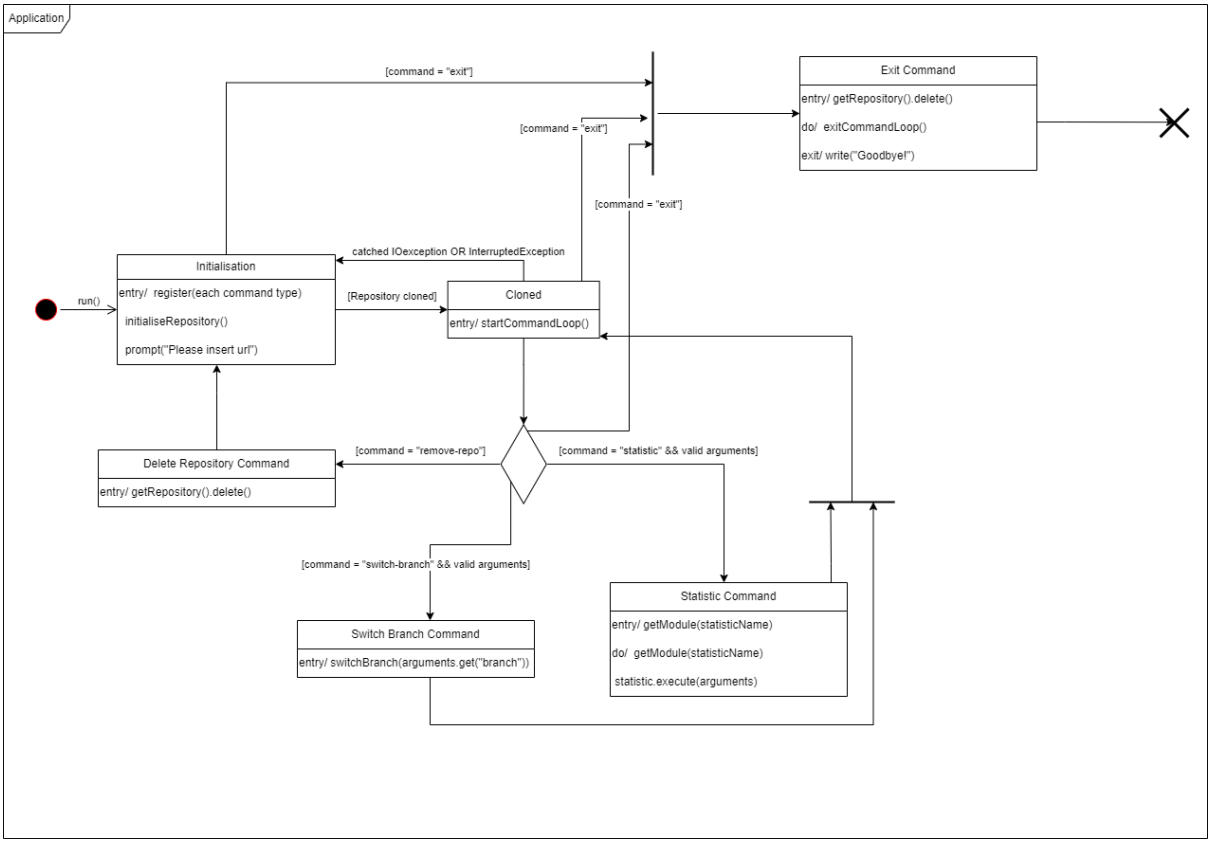
According to the changes made to the class diagram an instance of the class **CommandFramework**. On the other hand we have an abstract class **Command.** No objects can be created for that class. But an object is created for other related classes. To extend and further implement the abstract class five objects will be made of the classes **DeleteRepositoryCommand, StaticCommand, SwitchBranchCommand, HelpCommand, and exitCommand.** These five instances (extCmd, statCmd, sbCmd,

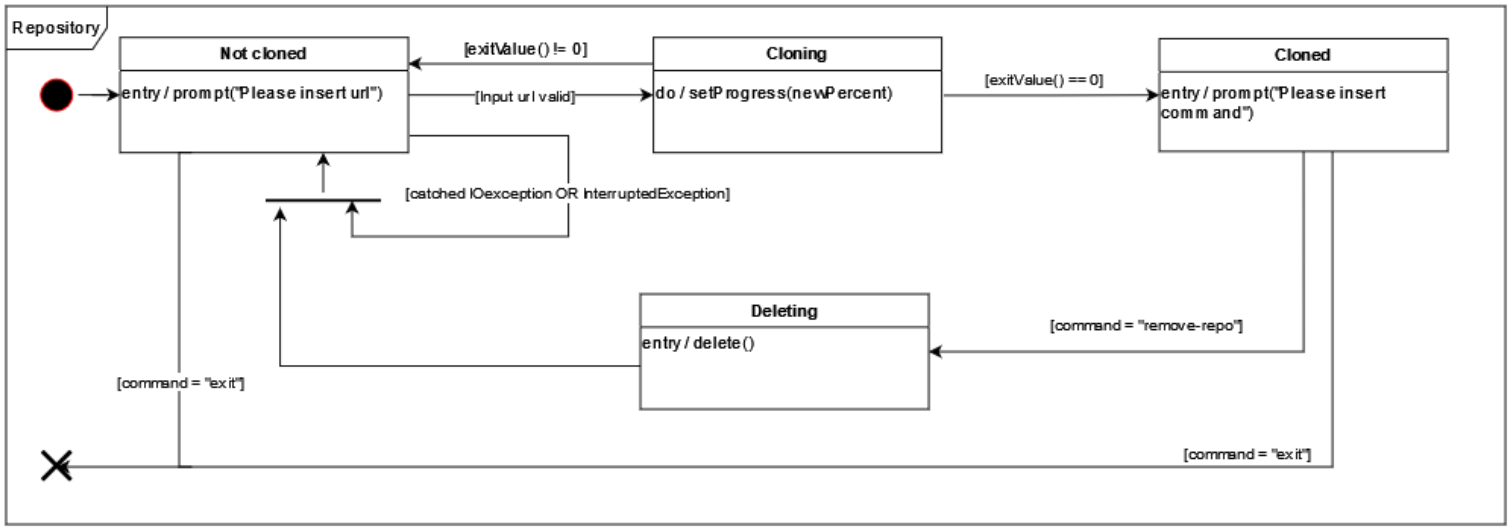
hlpCmd, and dltRepoCmd) will inherit from the **Command** class. For each command there will be one object of the **ArgumentParses,** as shown in the diagram. All improvements were done according to changes in the class diagram or the feedback of our TA Linus.

Revised state machine diagrams

No highlights can be provided in the textual description of the diagram, due to the complete overhaul of the description and making everything yellow seems annoying to read. The same story goes for the diagram.

Author(s): Márkó, Joachim





The models were fully remade apart from the **Repository** class's transitions and states, but commands (apart from remove repository) were removed and modelled in the **Application** class.

Most important changes:

- Diagrams have been untangled and split into 2 by duplicating the cloned states.
- Added the function calls as events instead of describing what the function does.

Events and guards in both models are described as method calls or as changes in variables, this way the flow of the program can be followed and the states can be understood. The states themselves are not coded in the program, therefore they function as abstractions to help distinguish between the main functions of the program and their interactions.

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 because there should not be a repository on disk the program can access. The user can only execute statistical commands if there is a local copy of a repository to work on, because the program cannot calculate anything without data. When the user provides a valid URL of a Github repository, the program creates a local copy of the repository. During cloning files from Github the class is in the Cloning state. After successfully cloning, we move to the Cloned state. If any network IO, interruption or git(hub) error will cancel the cloning process represented by the error arrow, put the program back in the Not Cloned state, instead of crashing the program to comply with QR2 (a common error should not kill the program).

From the Cloned state the user can now access the Statistic commands because there is something to calculate on now, as well as gaining access to removing the repository. This results in the deletion of the repository, after successful deletion the program goes back to the Not Cloned state. We are restricting the user access to the Statistic commands, in the Not Cloned state, to decrease the amount of errors made by the user. The program can be exited both from the Cloned and the Not Cloned state.

Application State Machine

The initial state is the Cloned state. From this state the application can prompt for a command input. **TerminalIO** handles the input then the **Application** class decides which **Command** class should be used: **SwitchBranchCommand**, **DeleteRepositoryCommand** or **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 to allow the user to input more commands. The **DeleteRepositoryCommand** goes to the Deleting state, and deletes the local copy of the repository which then returns to the Not Cloned state to restrict the user from using Statistic commands again and define these user errors out of existence. 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.

Revised sequence diagrams

Author(s): Joachim
Maximum number of pages for this section: 3

No highlights can be provided in the textual description of the diagram, due to the complete overhaul of the description and making everything yellow seems annoying to read. The same story goes for the diagram.

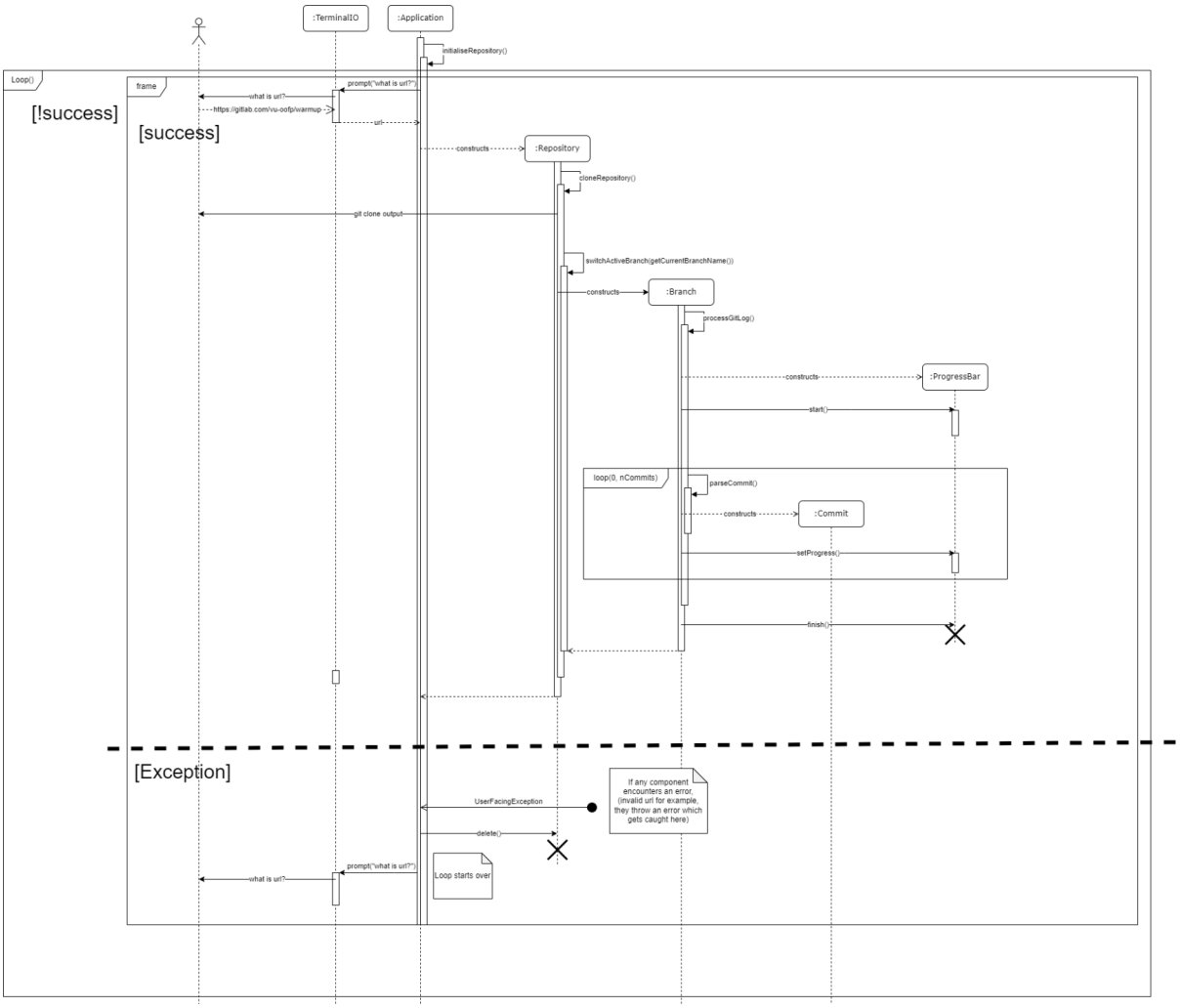
Why the changes in the diagrams?

The diagrams were a mess, and as Linus said: I cannot see a reality where these diagrams would be useful to me as a developer. This is why my first effort was to clean them up. I rebuilt them completely from scratch, and added much more space between the events and the classes. I added this space for the activation bars, that further improve the look of the diagram, and really show the law of demeter being used in our program. I omitted all return arrows except for the ones that actually return something.

The second piece of feedback was my misuse of the ref statement. So I removed it in all but one diagram.

The third piece of feedback was that my description was bad. So I destroyed it, and made a new one, focusing on why something happens in some class specifically. Thank you for reading!

Sequence diagrams cloning and switchBranch



As the program is started, some class is going to have to call Read() on TerminalIO to fetch the url to the github repository. This class must be **Application** because of the lack of any instances in the Repository module, in addition to the fact that no other classes should do this. This puts the **Application** class in the maestro role that we want it to have where it guides the program.

The user then inputs a url to the desired repository. No commands can be inputted because there is no repo on disk to run any commands on yet, which as said before defines these user errors out of existence. A **Repository** instance is then created by **Application** to represent the cloned repository, and pull all of the complexity that comes with interfacing with Git downwards into the repository module. The **Repository** first clones the files in its constructor then calls *switchActiveBranch()* on itself at the end of its constructor, this is instead of the **Application** calling *switchActiveBranch*(). Did you know the developers of the law of demeter once said: “if anyone is making an Application class that creates an instance of **Repository** the **Repository** should set itself up to prevent information leakage between the **Application** class and the **Repository** for setting up instances of **Repository**”?.

Before the cloning process a **ProgressBar** is instantiated to remove duplicate code between any processes that take longer than 3 seconds in our program and to comply with QR4. The progress is being updated in the **ProgressBar** to prevent the knowledge on

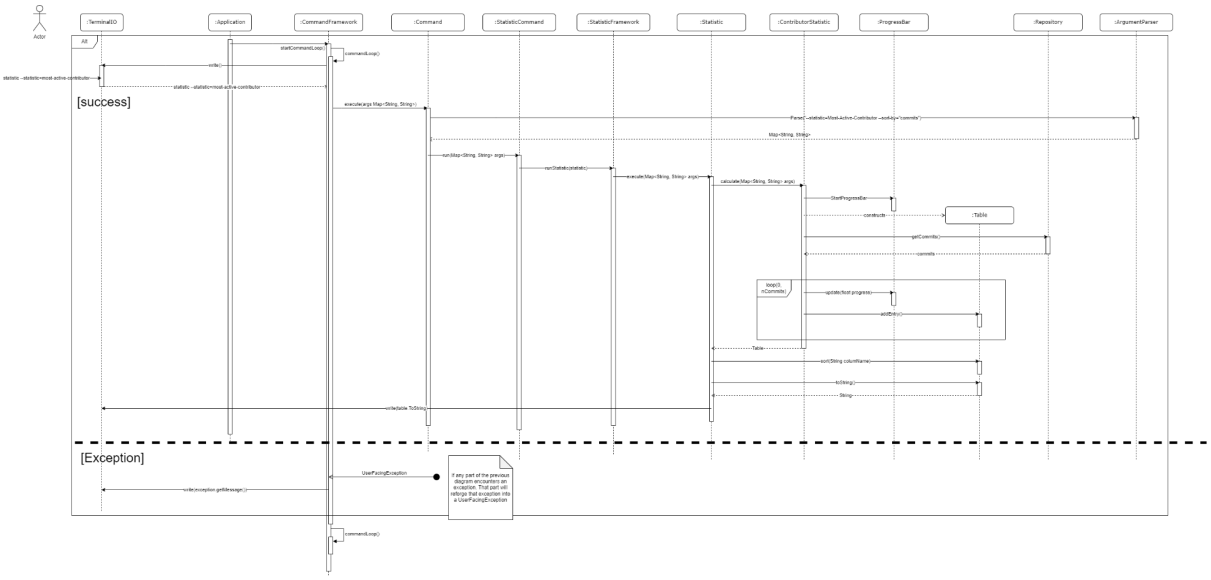
operating a progress bar from being leaked, and to minimise duplicate code. This update happens with *setProgress()* which you can see being used in our loop. We chose this approach for interfacing with **ProgressBar**, because of the easy state implementation, but more importantly the easy interface (only 2 argumentless methods and one method with primitive argument). 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 handler. If the cloning was not successful, we destroy the **Repository** instance and remove the files from disk which is done with the *delete()* method inside **Repository.** Why in the **Repository** class? Because this hides the usage of *filepath* and the file structure to other classes. After deletion, we ask the user to input the URL once again and try to clone.

When the clone is successful, the **Repository** class calls *switchBranch() which* immediately creates a default **Branch**, and populates it. This is done immediately to define possible user errors of forgetting to call git log or setup branches before inputting the first few commands out of existence. The *processGitLog()* method (again called from the constructor of the branch because the makers of the Demeter law also once said: "*processGitLog()* should be called from the constructor of the branch to prevent information leakage between **Repository** and **Branch**") executes the git log command and parses its output while maybe using the same **ProgressBar** class, if we can figure out a way to do this easily in the scope of the assignment, 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 a git log fails, we go back to the previous branch, because we know that branch exists, and we don't have to call git checkout. If this was the first branch to be parsed, we cannot go back to any previous branches, therefore we must delete the **Repository** instance, and go back to the beginning.

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

Statistics calculation in a sequence diagram

No highlights were provided for this diagram, because I built it completely from scratch.



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

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. The initiative lies pretty much always with the application class due to its maestro-like role and responsibility. The **Application** calls *promptForCommand*() on the **CommandFramework()**. *promptForCommand()* calls *read()* on **TerminalIO** to read the commands from the user. This happens in the **TerminalIO** to pull the complexity of reading from the command line downwards into a module (it's very verbose in java). This then reads the command from the terminal, which is parsed in the **Application** class.

The parsing previously happened in the application class, because it cannot happen at the command level because it's too late then. But now there exists the **CommandFramework** class that does this, because we felt like the **Application** class did not have too much to do with the command parsing and prompting, and it should therefore be pulled down into a module. Next to this, it also provides inversion of control to the **Command** subclasses to deduplicate code, and encourage generalisation of these classes.

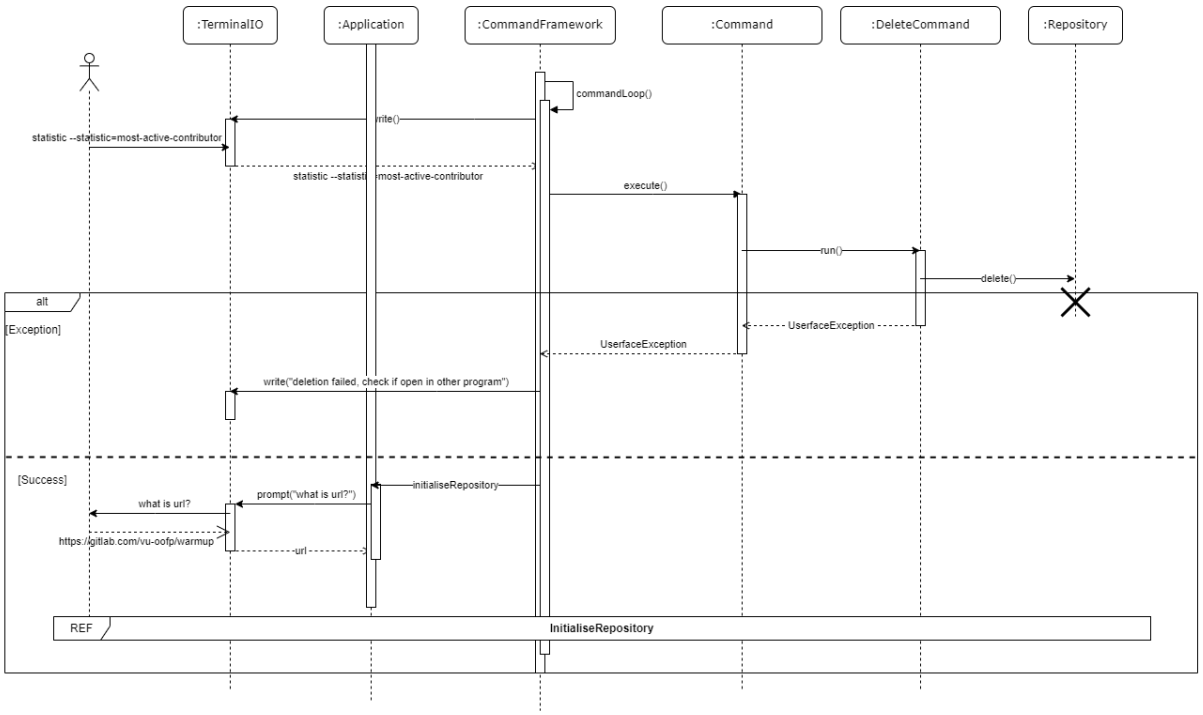
The **CommandFramework** also instantiates the right command for the job, and calls *Execute* on this command. *Execute* is a template method pattern in **Command** which calls upon the subclass modifiable **ArgumentParser**, and the subclass modifiable *run()* method. To parse arguments and use the command logic. This approach results in less duplicated code for the subclasses of **Command,** and therefore provides better generalisation for these as well compared to approaches with helper functions in super or distributed helpers.

Next, *run()* is called on the **StatisticCommand** by this template method with the parsed arguments, which invokes the wrath of the **StatisticFramework**. This framework registered all statistics at startup, and is responsible for parsing the name argument, and choosing the right statistic for the job. It calls the template method *execute()* on the correct statistic (in this case **MostActiveContributor**). This template method handles the sorting argument, and the printing of the table, while using the subclass modifiable *run()* method to actually make this table. This approach separates the statistics into different classes and makes the statistics more expandable, maintaining QR3. It also deduplicates code for table sorting, but allows new statistics to bypass this code by overriding the entire template method.

The statistic then provides this table and starts the **ProgressBar**. Then while calculating, it updates the progressbar to the user with the same interface to maintain QR4 (swift results). 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.

When anything fails, that class then throws a UserFaceException which is being written to the console by the **CommandFramework** because it can then also provide the exception handling framework, which is its job: providing frameworks.

Delete sequence diagram



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. By calling Delete() on the **Repository** class. This might be a shallow class, but it does hide The command framework from the **Repository** module, in addition to keeping the command infrastructure general which hard coded approaches just can't do. When deletion 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.

Implementation

Author(s): Everyone

The process of implementation

We designed our classes in such a way that all of them fitted in different modules that could be divided among developers. These developers then needed to know very little about each other's work and very little intermodule communication was required. This worked, developers would write all of their interface comments and method signatures on their separate git branches, then merge those branches into the assignment-3 branch when significant progress had been made. Zain set up branch protection and required reviews when merging to assignment-3 branch to keep our work in check. The work distribution was as follows:

- Repository module - Joachim
- UI module - Ammar
- ArgumentParser: Marko
- Command module - Zain with help with Marko
- Statistics module - Zain

Of course, plans are made to be broken, but this distribution lasted all the way until 3 days before the deadline.

Immediately, on day 1, a conventions document was made telling developers a javadoc implementation comment was necessary on every method and class, how all the git branches were set up, and how to work with those branches, etc. This immediately set the record straight and everyone in line.

Most important features of the implementation

Repository module

The repository module's most important feature is of course interfacing with git. This has been implemented with a **ProcessBuilder**. This is a standard java feature that can make a process to execute, from which you can get the stdout as a string, or you can just redirect it to your own stdout. Same story for the error stream. This makes it easy to clone the repository, get the git log output, and other methods like *getBranchNames*() returning all branch names.

UI module

The most important part of the UI module is of course the **Table** class which allows sorting. It's this sorting which is complicated. The table values are stored using a **List**<**List**<**String**>>, which we make an anonymous comparator for with lambdas. This looks complicated, but it really is just the comparator design pattern with some bells and whistles like error checking and anonymous functions.

Statistic module

The hardest part about the statistics is keeping it general. We therefore use a lot of inheritance and frameworking. This module calculates on data from the **Repository** class and uses **TerminalIO** class's **Table** class to organise data in a way that can be put to the

command line. Each statistic has its own class in which the calculation and table organisation takes place.

Command module

This module and the **CommandFramework** was made to assist the swift execution of commands in the **Application** class. In that class a command loop is started to handle text commands from the command line, the **Command** module and its classes mainly use **Repository** class's functions that were made to provide commands the functionality that they are used for.

Locations of important files in the repository

- Fat jar: build/libs/software-design-vu-2020-1.0-SNAPSHOT.jar
  Make sure to move the fat jar into a different folder outside the repository to execute it
- Main class: src/main/java/softwaredesign/Main.java
- Video demo: docs/demo.mp4, or https://youtu.be/4JIOnnxjYu8

Time logs

Team number	52		
Member	**Activity**	**Week number**	**Hours**
Zain	Working together	6	3.25
Marko	Working together	6	3
Joachim	Working together	6	4.5
Zain	TA meeting	6	1
Marko	TA meeting	6	1
Joachim	TA meeting	6	1
Ammar	TA meeting	6	1
Joachim	Git parsing	6	4
Marko	Command module, ArgParser	7	3
Zain	Statistics class, command framework	7	4.5
Joachim	Git parsing	7	0
Joachim	rewriting the assignment 2 doc	8	6
Zain	Command framework, statistics framework, Contributor and Weekday, SonarLint, fixes to other classes	8	18.5
Zain	Meeting with Linus	8	0.5
Zain	SwitchBranch, Exit, Help commands	8	5.5
Joachim	Working with Zain on implementation	8	7.5
Joachim	finishing implementation, making revised class diagram	8	6.5
Joachim	rebuilt all sequence diagrams and doc	8	5
Marko	State machine revision	8	5
Marko	Assignment 3	8	1
Zain	Class diagram, various things on codebase, finalising assignment 3	9	8.5
Joachim	Assignment 3	9	8
Marko	Assignment 3	9	1.5
		TOTAL	99.75