VU VRIJE UNIVERSITEIT AMSTERDAM

Software Design - Assignment 3

Team Number: 25, Mentor: Linus Wagner

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

Contents

1 Summary of Changes from Assignment 2 **1**

2 Revised Class Diagram **2**
2.1 Main Classes . . . 2
2.2 Utility Classes . . . 5
2.3 Data Types . . . 5

3 Application of Design Patterns **6**
3.1 DP1: Singleton . . . 6
3.2 DP2: Template Method . . . 7
3.3 DP3: Observer . . . 7
3.4 DP4: Adapter . . . 8
3.5 Evaluation of Other Design Patterns . . . 9

4 Revised Object Diagram **10**

5 Revised State Machine Diagrams **11**
5.1 Repository class . . . 11
5.2 Account class . . . 12

6 Revised Sequence Diagram **14**
6.1 Account Options . . . 14
6.2 Repository Options . . . 16

7 Implementation **17**
7.1 Strategy: from UML to Code . . . 17
7.2 Key Solutions . . . 17
7.3 Compliance to Basic Rules by Ousterhout and Robillard . . . 18
7.4 Notable Locations . . . 19
7.5 Demonstration of System . . . 19

8 Time Logs **21**

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

In the required color annotations of the diagrams, green is used for additions, orange is used for changes, and blue is used for design patterns.

1 Summary of Changes from Assignment 2

Author: Laura

Lack of Critical Reasoning The most notable point of improvement from Assignment 2 is in the area of describing our class, object, state machine, and sequence diagrams. Our team focused too heavily on the creation of complete diagrams and disregarded the importance of documenting the choices we made on the way. One major aspect that was missing from our previous descriptions was the consideration of alternatives and critical thinking. To rectify this shortcoming, we have focused more intently on our design-making process and have given special attention to ensure that our descriptions are thorough and comprehensive. By doing so, we address the issue of incomplete descriptions and have produced a more robust and well-rounded report.

Missing Functionalities in Class Diagram In our last assignment, our class diagram was missing the representation of some functionalities, namely the use of Exceptions and the methods for saving and loading accounts. Thus, we have explicitly modeled exceptions that are used by the classes to the model and included the missing methods with the **FileManager** class. A more thorough description of the changes can be found in Section 2. Additions to the class diagram are highlighted in green.

Lack of Clarification of Single vs. Multiple Data The feedback taught us that the separation of **SingleData** and **MultipleData** was not evident. For this reason, we have dedicated a paragraph in 2.1 to clarify the difference between both and the reasoning behind this implementation choice.

Unnecessary Final State in State Machine In the previous version of our State Machine Diagram for the Repository Class, we represented a final state that is entered by a transition out of ***MetricsExtracted***. We agree with Linus' feedback that this state is unnecessary because it cannot be reached; the composite state ***MetricsExtracted*** is not exited unless the guard *[choice==quit]* becomes true. Therefore, we have removed the unreachable final state as well as the transition leading to it.

Missing Error Handling in Sequence Diagram In the Sequence Diagram for our Account Options we have excluded error handling for wrong user input. However, this is not in line with our quality requirements Q1 and Q3, as mentioned in the assignment feedback. To fix this inconsistency, error handling has been included in this iteration of the Sequence Diagram (see Section 6.1).

2 Revised Class Diagram

Author: Lennart

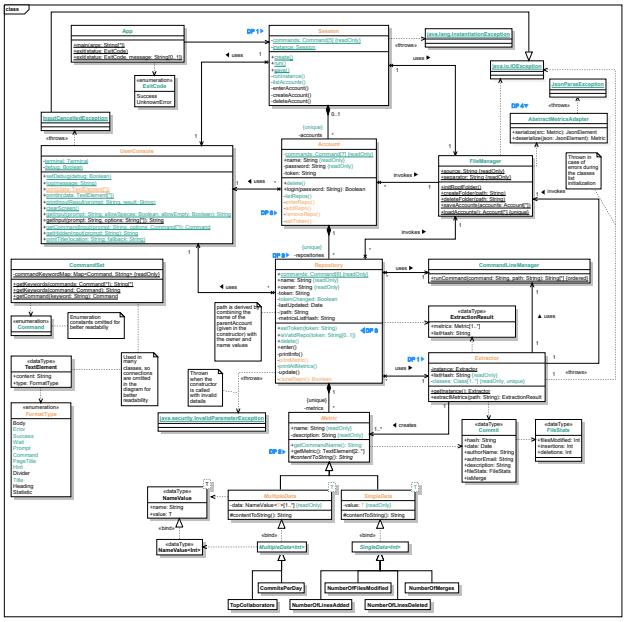


Figure 1: Class Diagram

Note: In this section, many design decisions are highlighted and briefly explained. Due to the page limit in combination with the amount of content we try to convey here, we decided to not explicitly name and explain alternative solutions everywhere. It should be obvious by the given motivation for the taken choices which alternatives were considered. For some more complex design decisions, there are still alternative solutions mentioned.

The following presents descriptions of the notable changes in the class diagram since the last report. To keep the focus of this report on the relevant parts, minor changes like changing visibilities or declaring attributes as constants (readOnly) are not mentioned or elaborated on. Most of the changes are due to some minor inconsistencies or inconsiderations in the first phase of the project that became clear once the implementation phase started. Classes and data types that remained unchanged from the previous model are not mentioned again. Please refer to the previous report for more information about those parts of the system.

The class **MessageSet** is part of the implementation but purposefully not part of this model. This is due to the class only providing subclasses that group individual I/O messages and not offering any structurally important functionality. The decision to use a dedicated class for this was taken to decouple representation and implementation of the functionality, such that changing the explicit I/O messages would need no modification beyond those in the **MessageSet** class. This approach also facilitates the intuitive design of the interaction messages by allowing one to see all messages in one place.

Some overloaded versions of methods as well as many helper functions and some helper classes are not modeled here to keep the focus on the essential parts and interfaces of the system. Furthermore, constructors are not modeled as their signature can be derived from the structure of each of the classes (e.g., an account is created by passing it a name and password).

One notable change from the previous version is that significant exceptions which might be thrown by classes are now explicitly modeled. This is addressing feedback from the previous assignment. Another minor change is the renaming of the “invokes” associations with the **UserConsole** class. Thereby, the diagram is now consistent such that all associations that might include a two-way exchange of data (so an invocation and a return value) are labeled “uses” and all associations that only have an invocation but no return value are labeled “invokes”.

2.1 Main Classes

App In the initial class design, the **App** class was envisioned to provide the entry point into the application, offer the main user interaction loop, and hold the state of the current session in the application. During

the implementation phase, it became apparent that the **App** class was responsible for too many different things. It did not only provide the "home" user interaction loop but was also used for handling the application exit with proper error codes as well as parsing command line arguments for setting up the console. Additionally, due to the need to have a static main method, in the initial implementation, the whole class, and thereby also the account list, was static. As the list clearly represents some sort of state, a static implementation did not seem appropriate upon closer inspection.

One approach to mitigating the static/non-static issues would have been to make the class have a static part that instantiates a non-static part to contain the accounts and offer user interaction. This approach, however, added significant complexity. So, to have a proper separation of concerns, the approach was discarded. Instead, everything related to the active session of the application has been refactored into the **Session** class.

Thereby, **App** is now a purely static class. It offers a *main()* method that creates and runs a new **session** as well as a public *exit()* method for controlled program termination. The *exit()* method is responsible for invoking *save()* on the active **session** such that the current session state is saved and can be restored at the next execution.

Session As explained above, the **Session** class represents an active session of the application. Such a session includes a list of accounts and runs an interaction loop for the user to access the application's functionality.

An instance of the class can only be created using the *create()* method and the created instance is not publicly accessible but can only be controlled via the *run()*, for entering the main loop, and *save()* methods. Thereby, the **Session** class can internally manage how to handle its instance and clients can't create more than one instance in any way. Consequently, as only one instance can be active, if *save()* is called, the savefile will be consistent with all (so just the one) instances of **Session**.

The (private) *runInstance()* method (invoked by the public *run()* method) allows user interaction to start and only returns if the user selects to quit from the home screen. *commands* specifies all the commands that are available in the user interaction (the same applies to the *commands* attributes of the **Account** and **Repository** classes).

The *accounts* attribute is modeled as a simple set even though the implementation uses a map to facilitate efficient lookup. Initially, the map was explicitly modeled, but it was determined that this reduced the readability of the diagram by cluttering the class without adding any notable benefit. The same holds for the set of repositories in the **Account** class.

Account The initial specification of the **Account** class proved to be largely accurate for the actual implementation. However, some functionality has been added.

Most notably, as the application uses a working directory in the file system, the *delete()* is used to delete all files belonging to an account (and its repositories) before an account is removed from the application.

Furthermore, methods for setting the token as well as listing all available repositories have been added, as the implementation showed that these are needed to provide satisfactory user interaction.

Repository Similarly to the **Account** class, the initial specification of the **Repository** showed to be largely appropriate for the implementation and was merely extended to accommodate some implementation-specific requirements.

The initial class attributes have been complemented by a Boolean value for saving whether the token has changed since the last update, such that the **repository** does the needed modifications at the next update. Apart from that, similarly to the **Session** and **Account** classes, a set of commands has also been added. In the implementation, the modeled *path* attribute is actually split into 3 parts (for the parent, repository, and combined path), but this has been abstracted in the model to improve readability while still showing that some path attribute(s) exist.

As described in Section 3.3, every **repository** instance is in an observer relationship with its parent **account**. More specifically, when an account changes its token, all **repositories** are queried with the new token for validation and updated upon success. The *isValidRepo()* and *setToken()* methods are invoked by the parent **account** in this context. *isValidRepo()* is furthermore used to validate the given details in the constructor. A detailed explanation of the problem, solution, and intention of the chosen approach is described in section 3.3.

The set of user-invocable methods has been extended by a *printAllMetrics()* method as this functionality was added after the initial design phase. Furthermore, the *getMetric()* method no longer takes an attribute, as it handles getting user input internally.

As each **repository** creates a folder in the filesystem, it also provides a *delete()* method to delete this folder with all its files before it is removed.

Extractor In the initial model, the list of available metrics was part of the **Extractor** as a list of strings representing the names. During the implementation, it became clear that the list should save classes instead. Thereby, during the initialization of the **Extractor**, the class types are saved and when metrics are to be extracted each constructor can be called directly.

With this approach, loading the class files (which is error-prone and thus needs extensive error handling) only needs to be done once in the beginning. This, in combination with the Singleton design pattern used for the **Extractor** also ensures that during a specific execution of the program, the list of classes remains the same.

The initial idea was to have the list of metric names public such that **repositories** could query the **extractor** for which metrics are available. However, **repositories** can simply use the *getCommandName()* method for each of the **Metric** instances they receive. Thereby keeping a list of metric names is no longer needed, especially as comparisons of whether a **repository** has metrics from an up-to-date list are done with the *listHash* anyways.

Henceforth, the *classes* list is now declared private as it is only needed by the **extractor** itself.

Metric During early implementation, it became apparent that each metric needs some identifier such that it can be selected by the user. One approach would be to use the name of the metric for this purpose, however, as the names can, and often do, contain spaces, using them as commands is inconvenient (as spaces would need to be escaped in the input due to limitations of the auto-completion in the UserConsole class). Another approach would be to add another attribute specifically specifying a command name for each metric. This approach, however, comes with two issues:

1. As the metric name and command are likely very similar, two attributes would be present that contain almost the same value, and changes to the name would also need changes to the command to keep consistency
2. The extractor checks whether the *name* of a metric matches the name of its class (ignoring spaces and capitalization). This, in combination with the requirement for Java classes in the same package to have distinct names, ensures that there will never be two metric types with the same name present. If the command could be set arbitrarily, two metric types could still have the same command leading to ambiguities.

The approach that was used in the implementation mitigates all these issues by automatically generating a command name from the *name* of the metric. The generated command name is the camelCase representation of the name of the metric and thereby well suited to be used as a command.

Single Data, Multiple Data Only small changes to the attribute visibilities were needed, as the necessity became apparent during implementation. As the distinction between the two classes was unclear in the previous report, the following provides a more specific explanation.

Any metric consists of a *name*, *description*, and some content that can be converted to a string by the *contentToString()* method. These three parts are combined by the *getMetric()* method defined in the abstract **Metric** class. As there are essentially only two different types of metrics (for now) for which every metric needs a similar *contentToString()* method, the abstract classes **SingleData** and **MultipleData** provide implementations of this method such that any specific metric class only needs to provide a *value* (for **SingleData**) or some *data* (for **MultipleData**) for the *getMetric()* method to properly return a textual representation of the metric. Thereby, it is apparent that **MultipleData** could not be substituted by an array or list of **SingleData**.

This approach reduces duplication of code, (e.g., for similar *contentToString()* implementations) to a minimum and offers a clean and easily extendable interface while also ensuring consistency in the representations of the metrics. In the case where a new type of metric would be added, simply a new abstract superclass similar to **SingleData** and **MultipleData** would need to be added which then provides a different kind of *data()* attribute and *contentToString()* implementation.

A notable change from the previous model is the use of a template parameter. Thereby, **SingleData** metrics are no longer required to use an integer as their value but can use any type and **MultipleData** can use NameValue pairs with values of any type (as long as the type is comparable). This was decided late in the implementation phase to facilitate even greater extendability of the metrics.

AbstractMetricsAdapter *Functionality explained in section 3.4.*

2.2 Utility Classes

All utility classes are classes with exclusively static methods and attributes and can't be instantiated.

UserConsole To avoid naming ambiguities with the Java **Console** that became apparent during early implementation the **Console** class was renamed to **UserConsole**.

While in the initial design phase, the class was modeled with a minimal interface representing the essential functionality, the new model shows the full interface as currently present in the implementation. As the **UserConsole** is the only interface between the user and the application, many different methods are needed to facilitate a seamless I/O interaction without having excessive complexity in classes that use the **UserConsole**.

Internally, the class uses a **TextStyle** class class to offer an abstraction of text styles and reduce complexity, however, this class is not modeled as its scope is limited to the **UserConsole** and it is thereby not relevant for understanding the structure of the application itself.

All methods do as the name would suggest, for more detailed information please refer to the documentation in the source code or the previous report. All methods that get user input only allow input that is part of the given set of options, thereby satisfying QR1.

CommandSet In the first steps of implementation, it became apparent that user-invocable commands need some sort of identifier. To enable effortless customization and facilitate the extension of the set of commands, it was decided to separate the internal identifiers of commands with their textual representation that will be used in the user interface.

The **CommandSet** holds all identifiers and textual representations and provides methods to convert from one to the other. Thereby, all classes that are involved with user interaction only need to have a set with identifiers of the available commands but do not need to be concerned with the representation. Adding a new command to the system is as simple as adding a new identifier and mapping it to a textual representation.

FileManager Many of the application's functions need some sort of file handling. This includes creating and deleting folders as well as loading from and saving to a savefile. This functionality is offered with some abstractions by the **FileManager** thereby removing complexity from the individual classes that need file interactions.

The **FileManager** handles OS-dependent path issues and offers information about the application's root folder path to clients. Furthermore, its methods like *createFolder()* or *deleteFolder()* interpret their *path* parameter relative to the application's root folder, such that client classes need not handle this themselves.

CommandLineManager Multiple functions of the GitHubMiner application require the execution of terminal/command line commands. As this is not a trivial task and especially the capturing of output generates some complexity, the **CommandLineManager** class was created to offer an abstraction for this functionality. The class offers a single method; namely, *runCommand()*. This method takes a command in the form of a string and a string specifying the path where the command shall be executed. Upon completion of the command's execution, the generated output is returned as a list of strings.

2.3 Data Types

Most of the enumerations present in the model are used by many classes. Therefore, many dependencies have not been modeled explicitly to avoid over-cluttering the diagram. Any class that has a call to the *exit()* method uses the **ExitCode** enumeration to convey information about the exit status. **Session**, **Account**, **Repository**, and **UserConsole** all use the **Command** enumeration to identify commands, and **TextElement** is used by most classes that are involved with I/O or textual representations of some form.

Commit, FileStats As the **Extractor** needs to convey information about a repository in the form of a list of commits for the metrics constructors, the **Commit** type was added. The data type simply groups all notable properties of a commit. To avoid having too many individual class attributes, all file-related attributes have further been abstracted into the **FileStats** type such that each **commit** simply includes a **fileStats** object for its file stats.

TextElement, FormatType During the implementation phase, the need for more format types became apparent, and thereby the **FormatType** enumeration was extended. Note that not all **FormatTypes** have a unique representation and some types are directly mapped to the default type in the console. The different types are still included to allow for further customization. The **TextElement** type however remained the same and proved to be useful in practice.

3 Application of Design Patterns

Author: Lennart and Laura

3.1 DP1: Singleton

3.1.1 Problem

In the GitHubMiner application, a so-called **Extractor** class is responsible for getting information from repositories and using this information to instantiate classes that extend the **Metric** class. For its functionality, the **Extractor** thus needs to have a list of all these classes. As in the GitHubMiner application, this list is extendable, loading of the list is done dynamically when an instance of the extractor is instantiated. As loading the list of metrics can have a significant performance impact on some less resourceful systems and all calls for extracting metrics should use the same list during an application run, it needs to be ensured that only one instance of the **Extractor** is instantiated and used during a run of the application. Furthermore, it should be possible to control when the class list is first loaded to facilitate error handling for unsuccessful load operations and manage the time point for the performance impact.

3.1.2 Solution

The above-mentioned issues are directly addressed by the Singleton design pattern. The nature of the design pattern ensures that no resources are wasted in loading the class list multiple times during an application run and it is guaranteed that every extraction uses the same list, as the list is loaded at first instantiation of the **Extractor** and cannot be changed later. With a lazy-evaluation implementation of the pattern, it is furthermore possible to control when the **extractor** instance is first created and thereby the class list loaded.

3.1.3 Intended Use

In the GitHubMiner application, the **Extractor** class is designed to adhere to the Singleton design pattern. Upon application start, the **instance** of the **Extractor** is created and it is checked whether loading the list was successful. In the case that the first invocation of *getInstance()* was unsuccessful, the application will quit with an error code. If the **instance** was instantiated successfully, the application enters its main loop where the user can interact with the system. From here on, any class can use the **instance** to access methods like *getMetrics()* or *getListHash()*.

3.1.4 Constraints

The design pattern does not impose any constraints on the system. An alternative to this design pattern would have been to use a static class, but as the extractor has some sort of a state (namely the list of metrics) having an instance of the extractor that represents this state is more fitting. With a static class, it would have furthermore not been possible to control the point of loading the list without adding more explicit methods for the process.

3.1.5 Further Usages

The **Session** shows a similar problem to the **Extractor**. Here also, at most one instance should exist at any time to avoid inconsistencies in the savefile (e.g., one instance saving its state, but another instance overwriting this state with its own). To ensure that this is true, the class is also designed in a Singleton-like way. Namely, *create()* creates a new instance if none is existing yet, and *run()* enters the main loop of the instance (if one exists). *save()* furthermore offers to save the current state of the active session (if one exists). The choice of never exposing the instance itself, but only offering control via selected methods was purposefully taken to simplify the interface. Clients of the **Session** class do not need to know about the internal structure of the class, but rather just need means to create, run, and save a session.

Even though the constraint of having at most one active instance of the class at any time could be managed entirely by the **App** class, as it is the only class to use the **Session**, fundamentally designing the class' structure around the constraint both decreases complexity for any client classes and reduces opportunities for developer errors in any extensions of the system.

3.2 DP2: Template Method

3.2.1 Problem

Every class that represents a metric needs to have a *getMetric()* method for returning a textual representation of the metric. To satisfy requirement F4b (Data Representation), all metrics should be printed in a similar structure, such that the user can easily concentrate on the data. Thereby, the *getMetric()* method should return a similar grouping of the title, description, and data. As there are several different metrics and the application is extendable with new metrics classes, there are two major issues present:

1. If every class has a largely similar *getMetric()* method, there will be significant repetition between the code of the individual get methods.
2. Designers of new metrics should not have the option for their metric to return a textual representation that is different from the standard set in the application.

3.2.2 Solution

The abstract **Metric** class is designed with template methods. It offers the final *getMetric()* method and in its implementation uses the *contentToString()* hook for the *data* as well as the *name* and *description* attributes of the metric to construct the textual description. Thereby, the general structure of the textual description will be the same for all metrics (namely name, description, and data). To furthermore unify the data representation for metrics of the same type (single vs multiple data) the **SingleData** and **MultipleData** provide a final implementation of the *contentToString()* method. This method uses the attributes of the class (value/data), which will be set by any subclass as hooks to include the data and thereby restricts any control over the representation of the data for the designer of the specific metric.

3.2.3 Intended Use

Any metric, when created, will set the *name* and *description* attributes of the **Metric** super-class and the *value* or *data* attribute of the **SingleData** or Multiple Data super-classes.

When at runtime a user wants to print a metric, the active **repository** will call the *getMetric()* method on the corresponding **metric**. This method is implemented by the abstract **Metric** super-class and uses the *contentToString()* method which again is implemented by the abstract **SingleMetric** or **MultipleMetric** classes (sub-classes of **Metric** but super-class of the specific metric). Together these methods will use the attributes set by the specific metric class (name, description, value/data) to form the textual representation of the metric.

3.2.4 Constraints

There is always a trade-off between room for customizability and a unified representation. In this case, the use of template methods greatly restricts the control over, and thereby the customizability of, the representation for metric designers. By extending one of the existing types (Single Data, Multiple Data) the representation of the metric is fixed and cannot be changed. However, as providing the data in a structured way such that the user can concentrate on the relevant information, it is more important that all metrics are represented in the same way (which was carefully designed by the original application developers).

The template pattern does not limit the extendability of metrics, as new types of metrics can always be added in the form of a new abstract class like **SingleData** or **MultipleData** that implements a new *contentToString()* method.

3.3 DP3: Observer

3.3.1 Problem

Instances of the **Repository** class need to know the access token of the account they are part of. However, they do not need to have any other information about, or access to, their parent account. The initial access token is given to every **repository** in the constructor of the class, which initially solves the issue. However, users of the application have the option to replace the access token of any **account** with a new one. This is necessary, as access tokens can expire and users should not have to re-add their repositories for a new access token.

3.3.2 Solution

To handle the information exchange between each **account** and its **repositories**, the **repositories** are observers of their associated **account**. When the **account** changes its access token, it notifies all its **repositories** about this change by calling the *setToken()* method for each **repository**. Thereby, **repositories** can decide how to handle the addition of a new token for themselves.

This approach loosely resembles the Observer design pattern, however, it includes some decisions that deviate from a "pure" Observer pattern. Apart from just notifying **repositories** about the change in the token, an **account** can also send a new token for testing of its validity. Furthermore, the **Account** class does not have explicit (public) *attach()* and *detach()* methods, but rather handles this functionality through the user-invoked *addRepo()* and *removeRepo()* methods. Added to that, there is no (abstract) **Observer** super-class which **Repository** could extend. This decision was taken to reduce complexity and improve the readability of the code, as by the inherent structure of the application, only **repositories** need to be notified about state changes of their associated access token.

3.3.3 Intended Use

When a user of the application adds a new repository to be tracked, the **account** creates the **repository** and thereby passes it the currently set access token. If the user now wants to change their access token, the **account** first queries all its **repositories** with the new token via the *isValidRepo()* method, to check whether all the previously added **repositories** are valid with the new token. If any of the queries returns false, the process is aborted. If that is not the case, then the **account** sets the new token and notifies all its **repositories** about the change, such that they can use the new token for upcoming updates.

3.3.4 Constraints

There are no notable constraints the design pattern imposes on the system.

3.4 DP4: Adapter

3.4.1 Problem

To be able to save the state of our system, we decided to store relevant information about the hierarchical class structures in JSON format whenever the user quits the application. This allows us to load the previous system state (including accounts, their added repositories, and relevant metrics) when the application is loaded again. To do this, we chose to use the serialization and deserialization functionality provided by the Gson library. However, we were met with a problem: when loading the JSON from the file to reconstruct the system's state, the JSON deserialization failed. The attempt to recreate our class structure (deserialization) relied on instantiating each object, however, our **Metrics** class is abstract - it cannot be instantiated.

3.4.2 Solution

The solution to dealing with the limitations of the Gson library's incompatible interface for abstract classes is to use concepts introduced by the Adapter design pattern. The Adapter design pattern allows us to override the default behavior of the JSON serializer and deserializer so that they are compatible with the abstract **Metrics** class. With the help of the Adapter design pattern, the reconstruction of our system's state from JSON succeeds. It should be noted that the use of the Adapter design pattern in our implementation deviates from the one presented in lectures: an existing class is not "wrapped" with a new interface while keeping the same idea of allowing two classes to work together that otherwise would not be able to because of incompatible interfaces. Our implementation uses an Adapter to act as a "middleman" between the **GsonBuilder** and the **JsonSerializer**<**Metric**> and **JsonDeserializer**<**Metric**> classes.

3.4.3 Intended Use

The Adapter design pattern has been accomplished by creating an **AbstractMetricsAdapter** class which implements **JsonSerializer**<**Metric**> and **JsonDeserializer**<**Metric**>. The class is used to override the *serialize()* and *deserialize()* methods on **Metric**. When the **GsonBuilder** is now instantiated (in the *initAccounts()* and *saveAccounts()* methods of **FileManager**), we call *registerTypeAdapter()* to register the **AbstractMetricsAdapter** to deal with the serialization and deserialization of the abstract **Metric** class.

3.4.4 Constraints

One constraint of this implementation of the Adapter design pattern is its lack of flexibility. If another abstract class were to be introduced into our code base, its deserialization would again fail. The solution would be to create an additional Adapter class that implements JsonSerializer and JsonDeserializer for that abstract class. This custom serializer and deserializer could then be registered to the **GsonBuilder** in the same way as the **AbstractMetricsAdapter** to solve the deserialization issue.

3.5 Evaluation of Other Design Patterns

Some other design patterns could have been used in the system but were ultimately disregarded, including but not limited to:

Flyweight Metrics of the same kind from different **repositories** that have the same value/data could refer to the same instance, as each instance of a metric can never change (it can merely be disregarded and replaced by a new instance). With this "constant" nature, each specific metric could have been designed as a flyweight pattern, such that essentially equal objects do not take memory space more than once. However, as it is unlikely for some metrics to be exactly equal between different repositories due to the large range of values they can take (e.g., the number of lines modified can easily be in the range of tens of thousands) the Flyweight design would rather add unnecessary complexity and performance overhead.

Command Even though the application uses commands to communicate the actions the user wants to perform to the currently active object, the Command design pattern does not offer any useful benefits. This is mainly due to the fact that commands do not carry any state or need to be offered in different ways, but rather are simple identifiers of possible actions in the application.

4 Revised Object Diagram

Author: Lennart

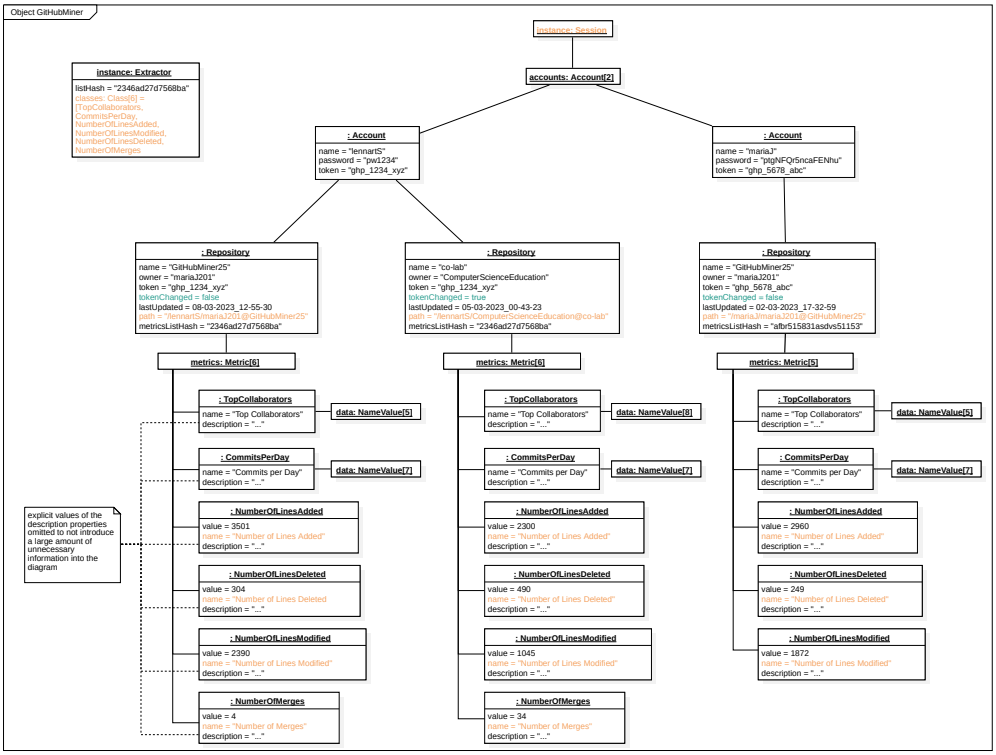


Figure 2: Object Diagram

The object diagram remained largely unchanged since the last report. This highlights the high similarities of the implementation with the first model of the system. There are only a few minor changes that also reflect the changes in the class diagram.

Note that no instance of the **AbstractMetricsAdapter** is modeled even though it can be instantiated. However, an object of the class is only created and used in the *loadAccounts()* and *saveAccounts()* methods and ceases to exist after loading or saving respective is done. As the current snapshot of the system is at a state where the accounts are already loaded, no instance of **AbstractMetricsAdapter** exists.

Extractor As already explained and reasoned about in section 2, the instance of the **Extractor** now uses a list of classes instead of a list of strings to represent the available metrics.

Repository One minor change is the addition of the *tokenChanged* attribute for each **repository**. This attribute is simply used to convey information about whether the token was updated since the last clone/pull and thereby the next pull requires a re-clone with the new token, upon which the attribute is set to 0 again.

The other notable change is the difference in the format of *path*. Initially, the path was structured as `accountName/owner/name`. This was done to ensure that even if an **account** adds two or more **repositories** with the same name (from different owners), no ambiguities in the folder structure arise. This, however, resulted in an unnecessarily deep nesting structure of the filesystem. Therefore, the path has been restructured to `accountName/owner@name`, where the second part is a single directory whose name includes both the owner and repository name. Thereby, the folder structure is one level shallower. The choice of the '@' symbol was intentional, as it is an allowed character for folder names on both Windows and macOS but is not an allowed character in the name of a repository on GitHub.

(The first idea was to use a '-' character, however, as '-' is a legal character in GitHub repository names, then in a situation with two repositories: "ab-cd"/"ef" and "ab"/"cd-ef" both of the repositories would get the same folder name, namely `ab-cd-ef`)

Metrics To comply with the restriction that the *name* of each metric needs to match the class name, some of the metric *name* attributes have changed.

5 Revised State Machine Diagrams

Author: Dovydas

5.1 Repository class

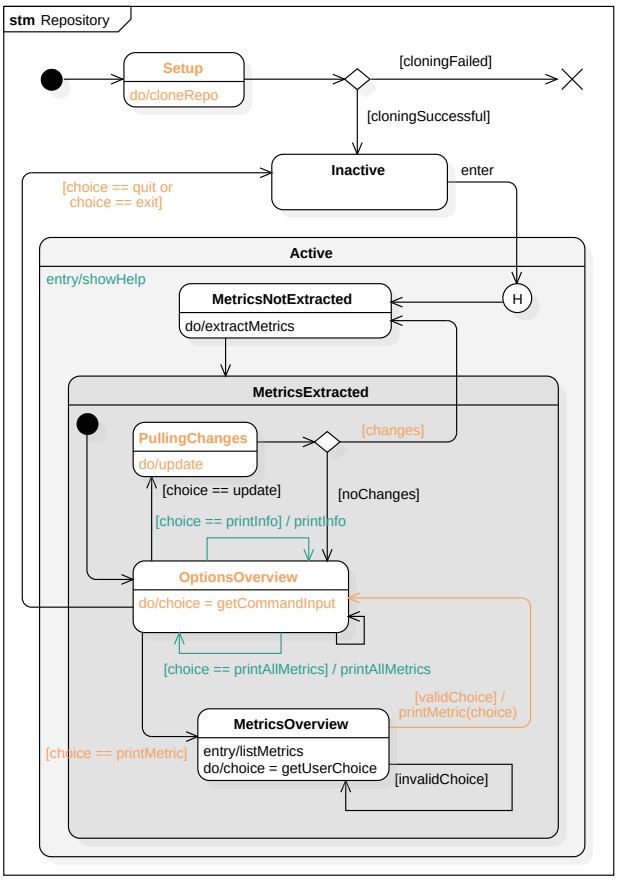


Figure 3: State Machine Diagram of the Repository Class

5.1.1 Changes

1. A guard for the transition from ***OptionsOverview*** state to ***Inactive*** state was extended with guard condition *[choice == quit]*. This covers the case when the user intends to quit the application, not only to exit the current repository which is depicted by transition with guard condition *[choice == exit]*. An alternative to this which we considered initially would be a separate transition from ***OptionsOverview*** state to the termination node as it would indicate the termination of all the parts that are connected to the current instance of the application. However, since the details of the repository and extracted metrics are saved locally, having to recreate the repository class for the same GitHub repository every time the application is restarted would create additional complexity to the state machine diagram, as firstly, there would have to be a check (with a decision node) whether the repository has been cloned before, and secondly, there should be a check once the ***Active*** state is reached whether the metrics were extracted before. Therefore, we decided to extend the aforementioned transition with the guard *[choice == quit]* which means that the repository class still exists even when the application is closed, but the repository and its details are stored locally. This makes the decision to include the history node more intuitive, as it goes to either ***MetricsNotExtracted*** state or ***MetricsExtracted*** state depending on the state before quitting the program, which avoids additional complexity in the state machine diagram.
2. A final state was removed from the ***Active*** state. Since there was no way of transitioning to the final state of ***MetricsExtracted*** state, the ***Active*** final state was never reached, thus it does not serve any purpose in the state machine diagram.

3. The event “deleteMetrics” was removed from the transition which goes from the inner decision node to ***MetricsNotExtracted*** state. Since our implementation of the “extractMetrics” method overwrites all existing metrics, there is no need to remove the current metrics beforehand.
4. Two self-transitions from ***OptionsOverview*** state were added which correspond to the options “printInfo” and “printAllMetrics”. Since this functionality was added to the implementation of the **Repository** class, it directly translates to the analogous changes in the state machine diagram.
5. A decision node that connects ***MetricsOverview*** and ***OptionsOverview*** was removed. An ability to quit the application during a selection of metric to print previously added complexity to the **Repository** class, thus we decided to not implement this option at all.
6. An ***Active*** state entry activity “showHelp” was added. In our implementation every time a repository is entered, a help message displays the possible actions on the current screen. We included it in the Repository state machine diagram to show that the Help page part of QR3 is satisfied in our project.
7. The names of the guards and events of the transitions were updated to correspond to the method names in the implementation.

5.2 Account class

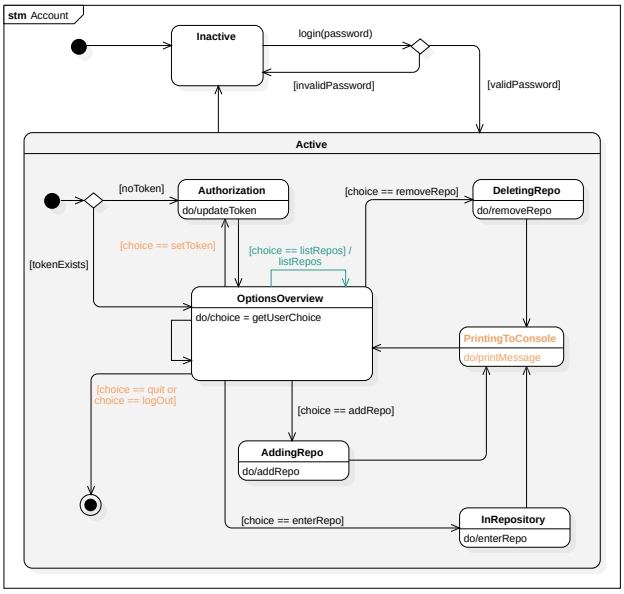


Figure 4: State Machine Diagram of the Account Class

5.2.1 Changes

The state machine diagram for **Account** class remained mostly the same, with some changes depicted below:

1. Decision node that checks the return value of activities “addRepo”, “enterRepo” and “removeRepo” was changed to the state ***PrintingToConsole***. Before the implementation phase, it was assumed that each of these activities would return an integer return value, and in the case the return value is non-zero, such a value would be translated into a particular error message. While this approach would be working, in the implementation, the integer error codes appear to be quite unclear and can lead to unexpected errors caused by typos when writing such numbers. In order to solve this, our implementation returns a custom message code (e.g. INVALID_REPO or REMOVE_UNSUCCESSFUL) which is translated to the detailed message in the **UserConsole** class. Since the resulting error messages are different for each activity, and the methods corresponding to the activities do not return any value (the success value would not be used anywhere) a single decision node cannot represent the current situation of handling the errors. In order to solve this, we changed the decision node to the state ***PrintingToConsole*** which calls the **UserConsole** class to print the message which indicates either a success or an error. Once the response is printed, the state transitions to the ***OptionsOverview***, where the user is prompted to perform the next option. An alternative to this solution would be an addition of a decision node which would be reached after the completion event of the ***AddingRepo***, ***DeletingRepo***, ***InRepository*** states. These decision nodes would check whether the activity has been completed successfully, and

if it would be not the case, then a transition with an activity printing the corresponding error code would be performed. When comparing the two options, the latter option would add considerably more elements to the state machine diagram (3 decision nodes and 2 transitions for each of the decision nodes with particular guards and activities), therefore, we decided to include the former design option with a single ***PrintingToConsole*** state in order to increase the simplicity of our state machine diagram.

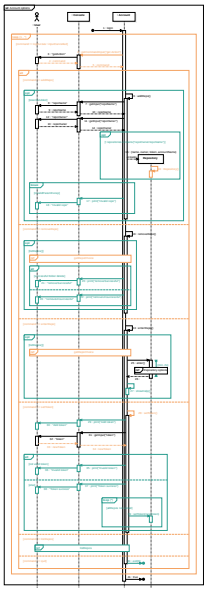
2. A self-transitions from ***OptionsOverview*** state was added which corresponds to the option "listRepos". The change indicates that we added this functionality was added to the implementation of the **Account** class.
3. Entry activity "listOptions" was removed from ***OptionsOverview*** state. This represents a change in the implementation: the list of the possible options is displayed when the TAB key is pressed, therefore the option is redundant.
4. The names of the guard "setToken" was updated to "updateToken" to correspond to the method names in the implementation.

While we also considered adding "showHelp" activity to ***Active*** state, it would complicate our system: if there is no token in the current Account, the "showHelp" function is called immediately after setting the token. This would require an addition of a decision node between states ***Authorization*** and ***OptionsOverview***. As we were prioritizing the clarity of the high-level state machine diagram, we made a decision to not include this activity.

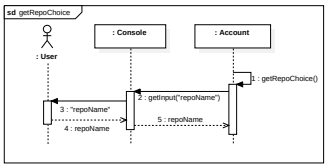
6 Revised Sequence Diagram

Author: Maria

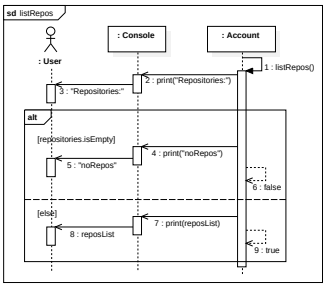
6.1 Account Options



(a) Account Options



(b) Get Repo Choice



(c) List Repos

Figure 5: Sequence Diagram Account Options

For this revised sequence diagram we show the interaction when the user logs in successfully to the account with an existing token. Firstly, we reconsidered the guard in the main loop from *login()* where it would stop iterating and exit the application when *[command == quit]*. However, we have changed the guard to be *[command != logOut]* so that it is possible to go back to the home page of the App instead of quitting the whole application. We have also added some additional checks with the optional fragment to make our system more robust, secure and minimize unexpected behaviour. This can be seen in the opt fragment where we check if a token is added before adding a repo, otherwise GitHub authentication will yield an error while the user is expecting the system to have handled this situation. It is also evidenced under the same command (addRepo) when we check if the name of the repository that is attempted to be added already exists in the list of repositories of our system, if it is the case the repo will not be added to avoid duplicate information that would yield undefined behaviour.

Taking into account one major point of feedback was the inclusion of error handling in the diagram, in accordance to QR1 and QR3, the InvalidParameterException thrown by the constructor of **Repository** for when a repository is not valid (determined with the help of the GitHub API) is included in this model with a break fragment. From our understanding there is no standard for exception handling in UML sequence diagrams and thus there are two main options that could be used to represent this. On the one hand, an alternative interaction can be used by using one of the operands for the "normal" execution of the program and a second operand that checks if an exception has been raised. On the other hand, the break interaction operator can be used, given it will check if the condition of the guard is met (in this specific example, if there is an InvalidParameterException) and execute the body of the operand. Afterwards the interaction continues in the next higher level fragment (the alt fragment checking which command has been provided). We have decided to choose the second option with the break operator considering that multiple nested interaction operators increase the complexity to understand the system.

There has been some renaming of functions and variable names to convey more meaning to what they perform and store, respectively. Taking as an example, *getInput()* has been changed to the more specific method name *getCommandInput()* and its return value has been renamed form *input* to *command*. *getCommandInput()* encapsulates *getInput()* by providing the list of options that the user is prompted with.

In our previous version of this sequence diagram we have shown that to create an instance of **Repository** we passed the parameters name, owner, and token; nevertheless, the accountName is now also passed as a parameter in order to create folders with the structure of `accountName/owner@name`. The alternative to the

current version was to not include the accountName in the file structure, but this would make it very hard to identify who owns each repository. Another alternative was to create a tree starting with accountName, under which folders for each repoOwners would be listed, followed by folders with the repoNames corresponding to a repoOwner. If considering this option, we would have 3 levels of folders in our file system, while on our current version we only need 2 levels given that we have merged the repoOwner and repoName in one level. In this way we can still differentiate between repositories that have the same name but are owned by different people, and minimize the complexity to delete empty folders when removing repositories from an **account**.

The *clone()* and *extractMetrics()* methods have been moved to be part of our other main sequence diagram (Repository Options), given that with the amount of detail included in the Account Options diagram, the sequence diagram was starting to be difficult to read. Besides, these methods are more related to the **Repository** options than the **Account** itself. Therefore, instead of showing specifically what methods are being called in the Repository class, the `message Repository()` is used to depict this and it is continued in the other sequence diagram.

removeRepo() and *enterRepo()* check the return value of the method *listRepos()* which is modeled as a ref to the sequence diagram listRepos (Fig. 5c). It has been included as an interaction reference so that the interaction can be reused by the command listRepos and at the same time to reduce complexity of this interaction.

There is an interaction reference to getRepoChoice (Fig. 5b) even though it only models 5 messages and is only used when removing and entering the repo; however, we have chosen to include it as a ref so that if at some point in the future we need to change the functionality of getRepoChoice, this will only need to be done in one place rather than in multiple places throughout our diagram. Additionally, this ref also increases readability of the sequence diagram Account options.

The interaction of removeRepo now includes if the deletion was successful or not, this to comply with our QR3 regarding error messages.

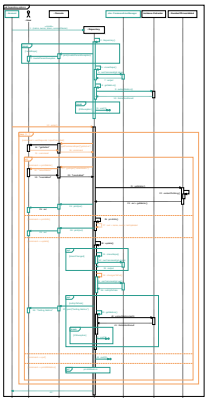
A duration constraint between the `message enter` from **Repository** to its reply message (return `message`) has been included to make it more clear that there is an elapsed time spent in this function before returning to **Account**'s login() main loop. As an alternative we considered including a maximum value of duration constraint, however, we discarded this option given that if the user is willing and has the resources, they can stay in the **Repository**'s main loop (*enter()*) for an indefinite amount of time. Furthermore, `message 27` (*showHelp()*) evidences the other part of our QR3 related to the provision of a help page so that the users know what the options are.

Token validation with GitHub API has been added to our model both for when the **Account** has just been entered and for each of the existing repositories related to the **Account** that had just updated the token. Furthermore, the communication from **Account** to each of the **Repository**s notifying that the token has been updated is also part of the diagram.

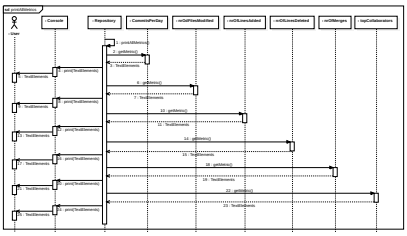
Given that a sequence diagram concentrates more in the interaction between objects and not within the object we have decided not to model certain operations in this diagram; for instance, the operations performed in the internal data structure stored in Account that saves the map with repositories.

Finally, the command to list all repositories is modeled in the interaction reference labeled listRepos (Fig. 5c) for the same reasons for the getRepoChoice sequence diagram to be included separately. It is worth noting that despite the implementation showing a few more exceptions thrown, namely for the InputCancelledException, in the implementation it is shown that no lines of code are executed when this exception is thrown because it will be equivalent to breaking the main loop (*login()*) and thus no break fragment to handle this exception is included but it is modeled as a guard of the loop.

6.2 Repository Options



(a) Repository Options



(b) Print All Metrics

Figure 6: Sequence Diagram Repository Options

This sequence diagram shows the interaction when the user enters the repo, while the metrics hash has not changed. This interaction is not included in this model as it will take place within the **Repository** object, and there will be no relevant message exchange with other instances of the system.

To begin with, as it has been mentioned before, the creation of **Repository** is now part of this model and not from Account Options due to its closer relation to the interactions in this diagram. Therefore, the new lifeline for this interaction partner is also included, which in turn makes it unnecessary to have lost and found messages for the *enter()* method and its return `message` respectively.

The constructor of **Repository** includes exception handling (related to QR3) with the same reasons to model it with a break fragment instead of an alternative interaction as mentioned before. Afterwards, *cloneRepo()* interacts with the new life line of the **CommandLineManager** class to run the command git clone. In contrast to the previous version of this diagram, when the metrics are attempted to be extracted, the instance of **Extractor** throws an IOException if the list file cannot be found or no classes from the list could be loaded, which exits the App with an UNKNOWN CODE(1).

One of our main changes from our previous version of this diagram is handling the printing from every alternative inside the option itself rather than assigning a common variable (out) that will after every iteration print its values. The main reason for this is that in our current version of the system there are some options that do not print at all while there are other that print to the Console multiple times; therefore, given the lack of commonality between the options, this interaction has been modified accordingly.

Regarding the update of the **Repository**, checks for the change in the token are first performed because if it is the case we would not be able to pull changes with the previous registered token and we need to *cloneRepo()* again. If after running the command git pull through the **CommandLineManager** there are updates recognized, the user is informed about the status of the system (QR3) and the metrics are extracted in a similar way as it has been explained before. An alternative to *cloneRepo()* when there is an existing version of the cloned repository is to delete current contents and then clone, but taking into account the git clone command overwrites an existing folder that has the same name than the repo being cloned, this alternative is superfluous.

Finally, we have decided to include yet another reference interaction (Fig. 6b) to show the interaction of the **Repository** with each and every class that extends from the abstract **Metric** class. This printAllMetrics sequence diagram shows the structure of a forEach loop of getting the metrics from the class and printing it to the user through the console. An alternative we considered for this representation of messages exchange is to only have an interaction partner for the **Metric** class, notwithstanding this is an abstract class and therefore cannot have an instance (for which interactions are being modeled) in a sequence diagram. This is the same reason why in the Repository Options sequence diagram we have also included the lifeline for the specific sub type of **Metric**, **NumberOfLinesAdded**. Likewise, even though it seems repetitive the exchange of messages from the **Repository** to *getMetric()* which would mean we could include a reference interaction to not repeat this interactions unnecessarily, we discarded this alternative because there is no common denominator of the metrics class and as argued above they need to be specific instance of **Metric**.

Note: as mentioned in the previous section (6.1), handling of the exception InputCancelledException is modeled as a guard that breaks the main loop of the **Repository** class (*enter()*).

7 Implementation

Author: Laura, Dovydas, and Lennart

7.1 Strategy: from UML to Code

The first step we took to migrate our implementation from its UML representation to actual code was the creation of a general skeleton, which was heavily inspired by the approach presented in UML@Classroom in Section 4.10. To lay these foundations, we relied heavily on our class diagram: each class in the diagram was translated to a single file in our project's `src` directory. We then filled in each of the methods listed in the class diagram into the corresponding classes, though the actual implementation of these methods was not yet established. Relationships that were displayed graphically by the class diagram were also included in the code skeleton, for example, the App-Account and Account-Repository relationships (noted by the +accounts and -repositories associations) were introduced into the code skeleton as Sets. It should be noted that the class diagram that we initially created for our skeleton underwent notable changes throughout our design process. More detail about changes to our class diagram is elaborated on in Section 2.

Once we established the general segregation between the different components of our system, we decided that the general flow of our system was the next important task to tackle. By the "general flow" of our system we refer to the interactions between our system's components. To model such interactions, we relied heavily on our Sequence diagrams. These gave us a good basis for establishing class relationships. The reason we chose to focus on this aspect of our project next was to ensure that testing of specific functionalities would also ensure that linked components of our system were still able to function successfully together. We could have alternatively started implementing each class' methods, though this seemed less ideal since it would not guarantee a seamless interaction between our classes once each independent functionality was implemented.

Task division was an important way for us to work on different components of the system simultaneously while aiming to reduce the number of (very) scary potential merge conflicts that could occur. Thus, each member of the team was assigned to implement functionalities of certain classes. Initially, the tasks were assigned as follows: Lennart worked mostly on the **UserConsole** class which deals with UI design. Maria focused on expanding the **Repository** class. Dovydas dealt with the extraction of the different metrics mostly in the extraction package. Lastly, Laura developed the saving and loading of the system state in the **FileManager** class.

Throughout the process of implementing the system's functionality, we continuously helped each other when something was unclear. Because we mainly worked together in the same physical space, conflicts were easily resolved and implementation uncertainties were discussed. The creation of the final code base required a lot of trial and error, googling strange Java behavior and documentation reading (especially for parts related to library use such as the JSON loading/saving), but we ultimately created an application that we are proud to present.

7.2 Key Solutions

7.2.1 Extendability of Metrics

GitHubMiner was designed with extendability in mind; namely, the set of available metrics is easily complemented with new metrics and even new types of metrics. If a new metric is to be added, only two actions are required:

1. The name of the metric needs to be added to `src/main/resources/metric_types.txt`.
2. A new class for the metric needs to be added the `src/main/java/softwaredesign/extraction/metrics` package. This class needs to extend one of the existing metric types (**SingleData** or **MultipleData**) and have a public constructor that takes a list of **Commits** and calls the super constructor with a name (which needs to be the same as the class name ignoring spaces and capitalization), description, and a value of type `t` (for **SingleData<T>**) or an array of `NameValue<T>` pairs (for **MultipleData<T>**) where `T` needs to be any type that implements comparable of `T`.

Template class skeletons for both a single-data and a multiple-data metric are included in `src/main/java/softwaredesign/extraction/metrics`.

If a new type of metric is to be added, a new abstract class similar to **SingleData** and **MultipleData** needs to be added to `src/main/java/softwaredesign/types/extraction`. The constructor of this class needs to call the super constructor with *name* and *description*. A template class skeleton for the metric type is also included as `src/main/java/softwaredesign/extraction/types/MetricTypeTemplate`. Please refer to the implementation of the existing metric types to gain a better understanding of the structure required.

An alternative way we could have allowed easy extendability of metrics would be to use a framework for dependency injection such as Spring, Guide, or Dagger as mentioned in the course slides. Dependency injection dynamically instantiates classes for you and injects them in relevant places in your code. This would remove the need to hardcode all metrics in the text file. Using this method, the code base would be reconfigured without requiring a recompilation. This approach would have made the extendability of metrics a lot more contained as there would be no need to hardcode instances in the text file, however, we, unfortunately, learned about this design principle too late into the process of our code creation so did not prioritize such a change. Still, we think that the current approach makes the metrics easily extendable and shows a creative way of dynamically loading classes at the program start including extensive error handling/prevention.

7.2.2 Extraction of the Metrics

Extraction of the metrics involved several stages of thinking. Firstly, we thought of some naive solutions: run `git log` command every time the repository is entered and iterate through the list of commits to calculate a requested metric. While this solution seems to be simpler in terms of implementation, it is both inelegant (contains a large amount of repeated code) and time-consuming (waiting for the calculation of the metric every time it is requested). To solve this, we decided to store the details locally: once a new repository is added to the account, it is cloned to the computer; similarly, once a repository is entered for the first time (or when the user asks to update the repository by pulling the changes), all existing metrics are extracted from the existing commits and saved to the JSON file. Having this in mind, getting the metrics during the user interaction with the system only involves reading and printing the requested metric from the JSON file, without any computational needs.

7.2.3 Loading and Saving the System State

As described in our functional requirement F5, software users should be able to store and reload data about their repositories and access token every time they reopen the application. To implement this functionality, we needed to create a way to load and save the system state to a file. We chose to store the system state in JSON format because it is familiar to us and offers good readability. To interact with our JSON file we use functions provided by the Gson library. Every time the application is reopened, the previous system state is restored by deserializing the information stored in the `data.json` file. When the user enters `quit` during the application's lifetime, this file is modified to match an updated system state by serializing the information about the current system state to JSON. It is notable that the exact location of the `data.json` file differs depending on the Operating System which runs the application (see Section 7.4). We chose to use these locations in an attempt to adhere to a standard; these locations are commonly used by other applications to store their own application data.

7.3 Compliance to Basic Rules by Ousterhout and Robillard

Throughout the project, we made sure that our classes complied with the basic rules by Ousterhout and Robillard. Firstly, we adopted the strategic programming mindset as suggested by Osterhout. In this mindset, a working code that just implements the requirements (tactical programming) is not enough, we invested some time to improve the current design of our system. With proactive investments, we thought about how different parts (Application, Account, Repository, Metrics) could interact with each other before starting the implementation of specific metrics. For instance, this investment allowed us to notice that a **Commit** should be modeled in its own class, improving the coordination between **Repository** and **Metrics** classes. With reactive investments, we were spending a bit of extra time thinking about how we could make the current implementation clearer. This involved adding some comments where needed, fixing SonarLint issues and warnings, and restructuring parts of the code after they were implemented.

Additionally, during the implementation phase, we aimed not to include a large number of small classes, as it creates additional complexity, for example, with more classes and interfaces, it becomes harder to track and find a desired component. An example of this is the class FileManager. We had an idea to have three separate classes related to the creation of directories, file paths, and JSON files, but since the functionality of these classes appeared to be very similar, we decided to put the related methods into a single class which reduces the separation and makes the hierarchy of the components simpler.

Moreover, several core methods (e.g. in classes **Account**, **App**, **Extractor**, **UserConsole**) include documentation that relates to the suggested interface documentation style. The comments contain information about parameters (with keyword @param), return values (@return), and a general description of the method that brings additional information in a different level of detail.

We adopted Robillard's suggestion to use enumerations to represent values that are part of a collection. For example, our Command enumeration inside of the **CommandSet** class holds a collection of all possible commands that a user may execute. The use of this enumeration allows for a clear distinction of which classes perform which functionalities, for example, the **Account** class holds a set of relevant Commands that it may perform. The alternative to using an enumeration in this situation would be to use integer codes. However, this approach is a lot less favorable as it drastically decreases the readability of the code and introduces unnecessary complexity.

Furthermore, the inclusion of the design patterns in Robillard's book helped us to understand their intended use cases, especially for the Singleton and Template design patterns, which were used as described in Section 3.

7.4 Notable Locations

Component	Location
Main Java Class	`.../GitHubMiner25/src/main/java/softwaredesign/App.java`
JAR File	`.../GitHubMiner25/build/libs/githubminer-by-pirates-1.0-SNAPSHOT.jar`
JSON File (Mac OS)	`.../Library/Application Support/GitHubMiner/data.json`
JSON File (Windows)	`...\AppData\Roaming\GitHubMiner\data.json`

7.5 Demonstration of System

A demonstration of our system can be found at: https://youtu.be/lgX12Jjb3Is. Here, the following flow is executed:

1. Start of Application: The application is started by running `make`. Upon initialization, the previous application state is loaded from the `data.json` file. Once the loading is complete, a title is displayed welcoming the user.
2. Account creation: An account is created with the username "Linus". An account keeps track of any repositories added by the user as well as the token needed to access private repositories linked to it, as specified by our F5 functional requirement. After successful creation, a directory with the account name is created.
3. Entering an account: The "Linus" account is entered. This exposes all repositories saved for the target account and allows metrics of a single repository to be accessed.
4. Adding a token: A token must be added for repository access to be allowed. The token allows the cloning of private repositories linked to the token's GitHub identity. This fulfills our F1 functional requirement.
5. Adding a repository: A repository must be added to an account before metrics can be accessed. The repository name and owner must be specified for the cloning to be successful, in this case "GitHubMiner25" and "mariaJ201". After the repository has been validated and cloned into a new directory "mariaJ201@GitHubMiner25", all metrics are extracted, complying with F2.
6. Entering a repository: The "GitHubMiner25" repository is entered. Now all relevant metrics related to the repository can be accessed.
7. Printing a single metric: A single metric can be displayed.
8. Printing all metrics: It is possible to also display all metrics of the "GitHubMiner25" repository.
9. Exiting a repository: Exiting a repository allows the creation and entering of a different repository.
10. Listing all repositories: Since there is only one repository "GitHubMiner25" linked to the account "Linus", it is the only one listed.
11. Removing a repository: A repository can be removed from the account. Additionally, it is removed from the application's save files.
12. Listing all repositories: After removing the "GitHubMiner25" repository, no more repositories are linked to the account "Linus".
13. Logging out an account: To enter a different account, we must first log out of the currently active one, in this case, "Linus".
14. Listing all accounts: Listing all accounts demonstrates that an account "Bob" is also saved in the application.
15. Deleting an account: Deleting the account "Linus" removes the corresponding directory "Linus" from the application's save files.

16. Listing all accounts: After deletion of the account "Linus", the set of accounts saved by the application also is updated.
17. Quitting the application: When the application is quit, a goodbye message is displayed. Then, the current system state is stored in a file before exiting.

8 Time Logs

Team number	25		
Member	**Activity**	**Week number**	**Hours**
Lennart	Define general concept + start making presentation	1	3
Laura			3
Maria			3
Dovydas			3
Lennart	Team contract + presentation preparation	2	2
Laura			2
Maria			2
Dovydas			2
Lennart	Test UML environment (diagrams.net), draft classes for class diagram, review feedback and update presentation accordingly	3	4
Laura			4
Maria			4
Dovydas			4
Lennart	Create drafts of class diagram, state diagram, and sequence diagram for review by mentor (together)	4	8
Laura			8
Maria			8
Dovydas			8
Lennart	Work on class, sequence, and state machine diagrams (together)	5	4
Laura			4
Maria			4
Dovydas			4
Lennart	Finalize class and create object diagram		3
Laura	Write up feedback points, prepare document, finalize state machine diagram		3
Maria	Create sequence machine diagram		3
Dovydas	Create state machine diagram		3
Lennart	Reviewing all diagrams, writing report		14
Laura			14
Maria			14
Dovydas			14
Lennart	Redistributing work + creating the code structure + implementing basic functionalities	6	28
Laura			18
Maria			22
Dovydas			20
Lennart	Fixing bugs + removing warnings	7	15
Laura			14
Maria			12
Dovydas			10
Lennart	Final code "cleanup" + diagram refinement + report writing	8	22
Laura			22
Maria			22
Dovydas			22
		TOTAL	379

Figure 7: Time Log