

KARLSRUHER INSTITUT FÜR TECHNOLOGIE

DESIGN DOCUMENT

Numerical Linear Algebra meets Machine Learning

Fabian Koffer

Simon Hanselmann

Yannick Funk

Dennis Leon Grötzinger

Anna Katharina Ricker

Supervisors

Hartwig Anzt Markus Götz

December 15, 2018

Contents

1	Overview	4
2	Module Interaction	4
2.1	Class Descriptions	4
2.1.1	Class CommandLineInterface	4
2.1.2	Interface OutputService	4
2.1.3	Interface Subscriber	4
2.1.4	Class Observable	4
2.2	Class CLIOutputService	4
2.2.1	Class Controller	5
2.2.2	Class CommandParser	5
2.2.3	Class Command	5
2.3	Activity Diagrams	5
2.3.1	User Input Activity Diagram	5
2.4	Class Diagrams	7
2.4.1	Command Parsing	7
2.5	Component Diagram	9
2.5.1	Model View Controller Component Diagram	9
2.6	Sequence Diagrams	11
2.6.1	Controller Sequence Diagram	11
2.6.2	Print Overriding String	13
3	Collector	15
3.1	Class Description	15
3.1.1	Class Collector	15
3.1.2	Class Saver	16
3.1.3	Class Generator	16
3.1.4	Class Ssget	16
3.1.5	Class Validator	16
3.1.6	Class CommandLineInterface	17
3.1.7	Class View	17
3.1.8	Interface OutputService	17
3.2	Sequence Diagrams	17
3.2.1	Interface Subscriber	17
3.2.2	Class Observable	17
3.2.3	Interface OutputService	17
3.3	Interface Subscriber	18
3.4	Class Observable	18
3.5	Class CLIOutputService	18
3.6	Class CLIOutputService	18
3.6.1	Class CommandParser	18

4	Labeling Module	19
4.1	Class Descriptions	22
4.1.1	Class LabelingModule	22
4.1.2	Class Solver	22
4.1.3	Class ConcreteSolver	23
4.1.4	Class preconditioner	23
4.1.5	Class Concretepreconditioner	23
4.2	Activity Diagrams	23
5	Training module	23
5.1	Class descriptions	25
5.1.1	Class Configuration File	25
5.1.2	Class TrainingModule	26
5.2	Activity Diagrams	27
6	Classifier	30
6.1	Class Classifier	30
6.2	Interface Solver	30
6.3	Class ConcreteSolver	30
6.4	Class Matrix	30
7	Classes which more than one Module uses	30
7.1	Class Loader	30
7.2	Class Validator	30
8	Glossary	32

1 Overview

2 Module Interaction

2.1 Class Descriptions

2.1.1 Class CommandLineInterface

The view represents the concrete command line interface. Therefore it only consists of two methods. The first one is readInput that receives a message that will be displayed and reads the next user input. The other method is createOutput. This method prints a string to the CLI.

2.1.2 Interface OutputService

The OutputService interfaces can be implemented and passed to a module to receive the output of the modules. Therefore it has methods that represent different ways output can be displayed.

2.1.3 Interface Subscriber

The Subscriber interface only provides the method update() which will be triggered by an Observable upon receiving new values.

2.1.4 Class Observable

The Observable class can be used to notify Subscribers when new values are provided. Subscribers can subscribe themselves to an Observer to be notified get notifications. The next() method calls update() on each subscriber.

2.2 Class CLIOutputService

This class implements the OutputService and the Subscriber interface. On creation it gets a reference of the View to which it will pass the lines the modules wants to output.

It also implements the Subscriber interface to subscribe itself to an observable. This can be used to display lines that are overwritten with new values like an progress bar or a counter.

2.2.1 Class Controller

The controller is the main entry point for the program execution. It creates the view, receives the user input, calls the parser to create a command from the input and starts the module the user wants.

2.2.2 Class CommandParser

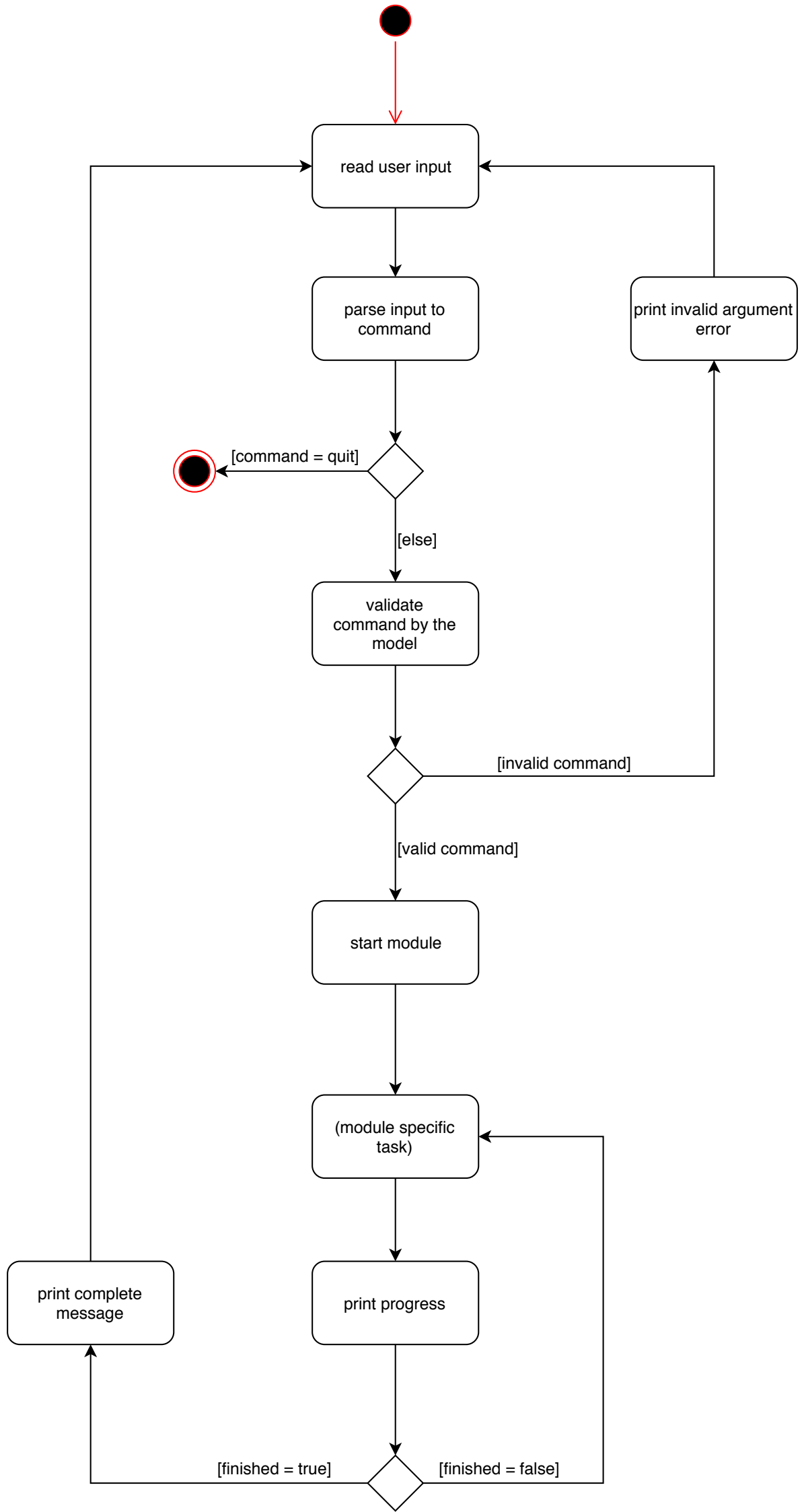
The CommandParser is a static class that gets the input which the user entered and parses it to a concrete command-object.

2.2.3 Class Command

The Command class holds all the information entered by the user that is needed to execute a module. There is one command subclass for each module and the command class also validates that all parameters are available to run the module. The command also has a execute method which runs the specific module with all the arguments it needs.

2.3 Activity Diagrams

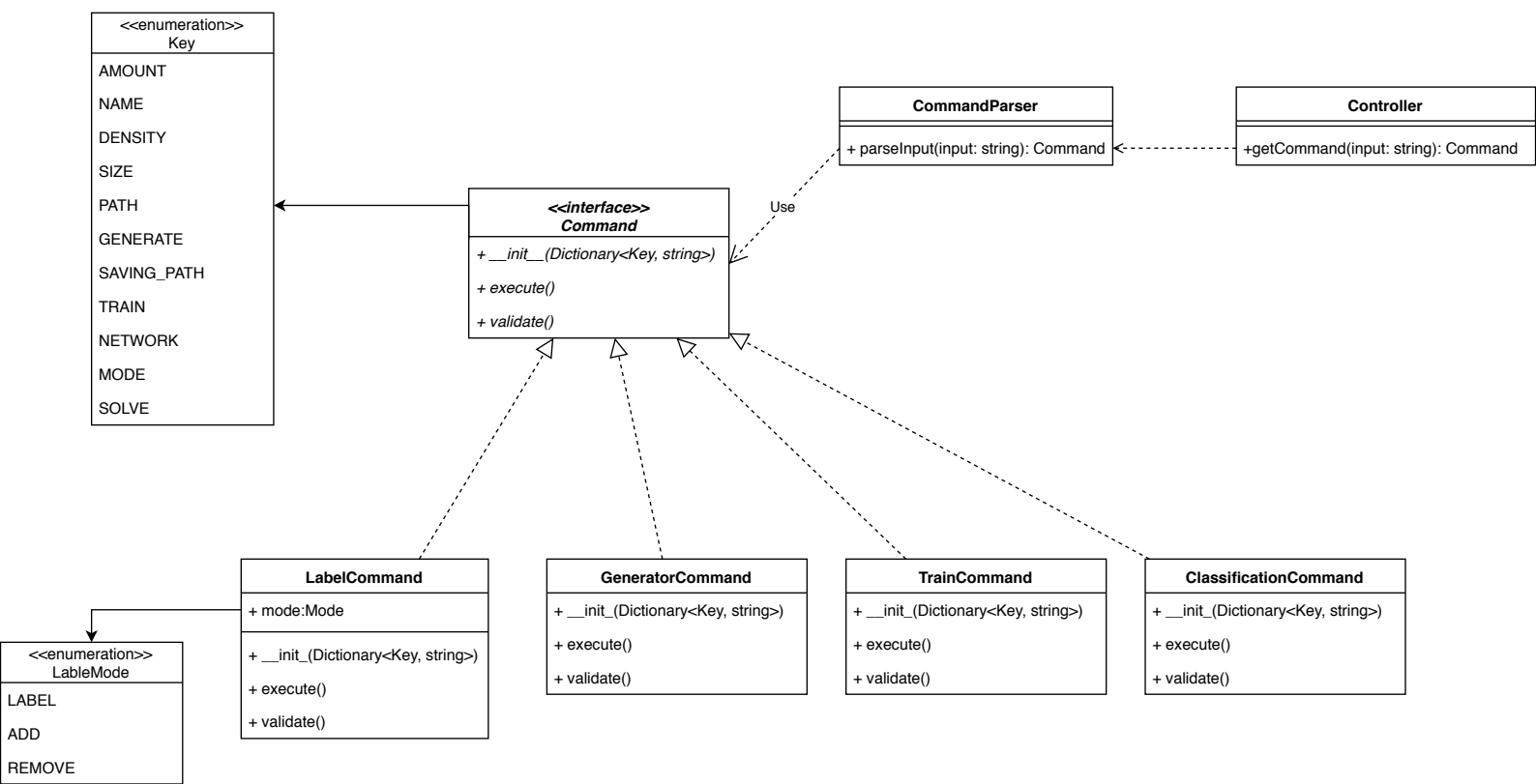
2.3.1 User Input Activity Diagram



This activity diagram shows the main workflow of the whole program. The program first waits for user input. After the user enters his input into the command line interface, the `CommandParser` parses this input to a command. If the command equals `quit`, the program will exit. If not the command will be validated to contain enough information to start the desired module. If it is not valid, an error message will be printed and the user can enter a new input. If the command is valid, the module will be started. Therefore the module computes his specific task in iterations. After each iteration it print the current progress to the user. This two activities repeat until all tasks are done. When the module is finished, the program prints a complete message and start waiting for new user input.

2.4 Class Diagrams

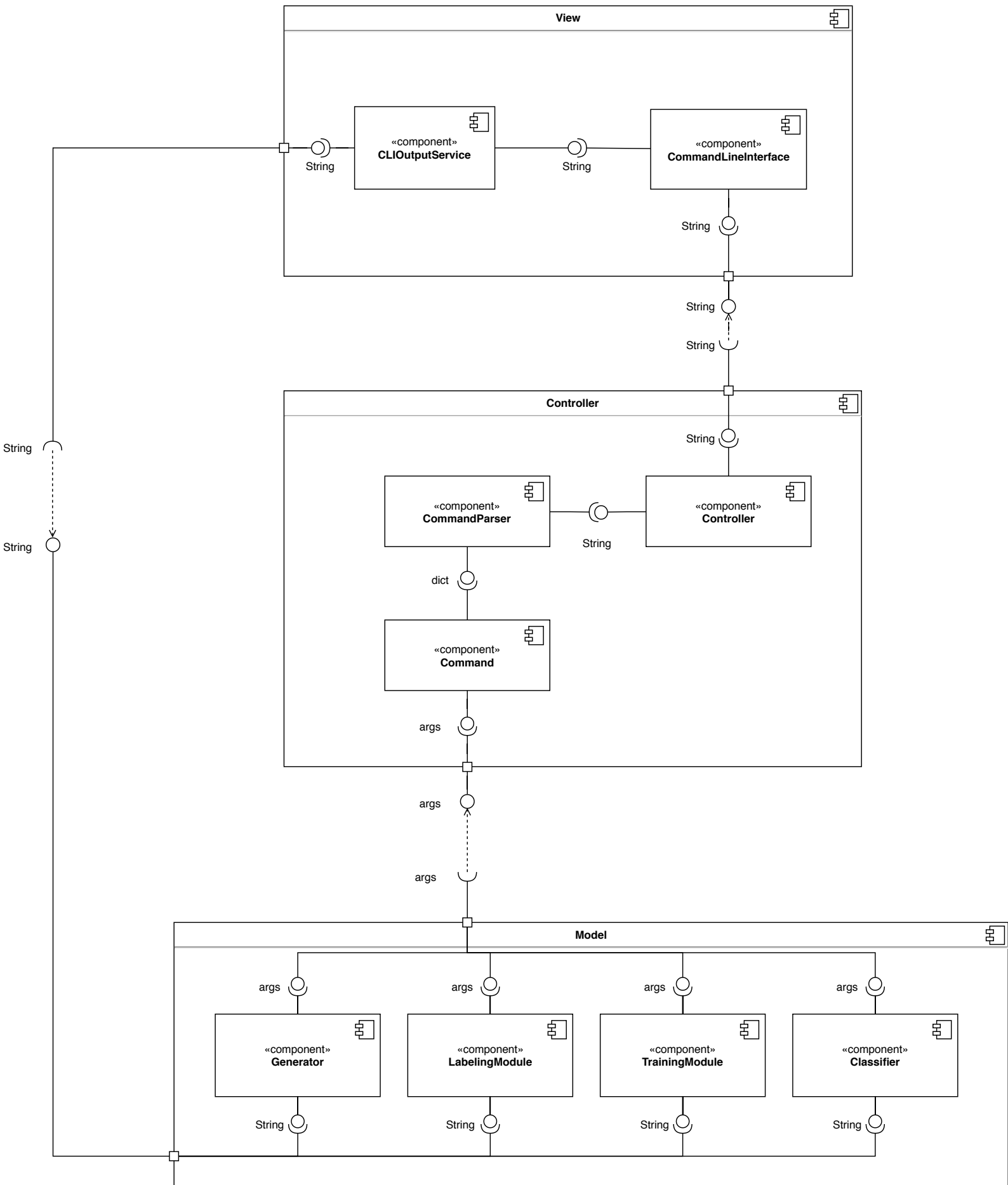
2.4.1 Command Parsing



This class diagrams shows all classes that are used by the CommandParser. The CommandParser has one method that receives an input string and returns a command. This method uses the first word of the input string to determine which module is supposed to be started. Based on this he creates an instance of this subclass of the command. The remaining string will be parsed to a dictionary with keys and values. This dictionary is passed to the constructor of the command object. In the constructor of the command, the command also calls the validate() method. This method checks if all arguments that are necessary for the specific module start are provided. After that is finished the command can be returned by the CommandParser. The execute method of the command can be then used to start the modules computation.

2.5 Component Diagram

2.5.1 Model View Controller Component Diagram



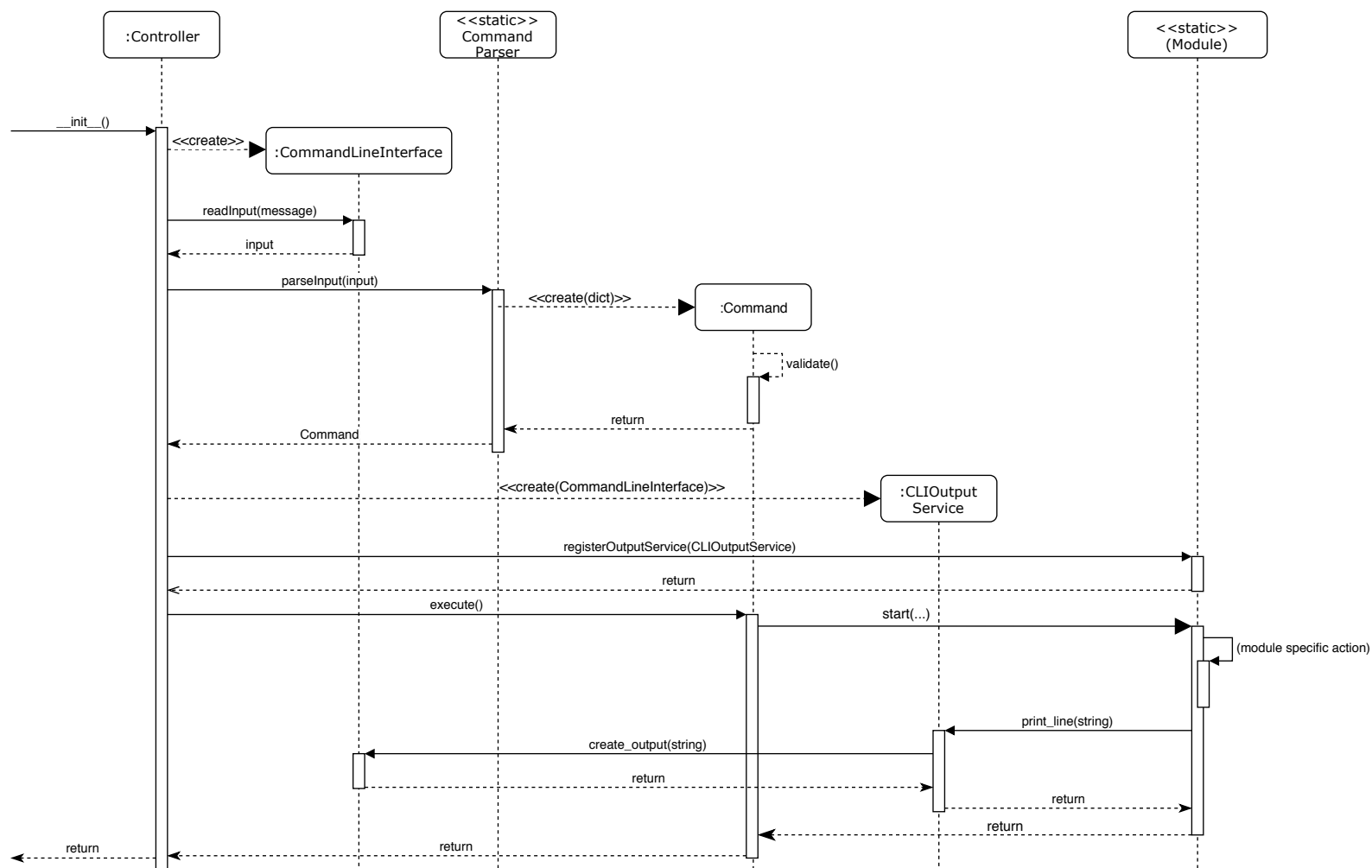
This component diagram shows the main interaction between the modules. For our main structure we use the model view controller. Each class belongs to one of the three parts: view, controller or model. In the view there is the `CommandLineInterface` which like the name suggests represents the command line. It receives string it displays and makes the user input available for other classes. The other class in the view is the `CLIOutputService`. This class works as the observer for the model and receives the strings the model wants to output. It then provides strings that can be displayed by the command line interface.

The next big part is the controller. This part's main component is the class that is also called controller. This class gets a string by the `CommandLineInterface`. This string is then passed to the `CommandParser`. This class creates the right command and passes the arguments to this command. The command stores these arguments and makes them available for the modules.

The model consists of the four static modules that make the computations. These modules get the arguments that are provided by the command. The modules can also provide strings that can be displayed in the view.

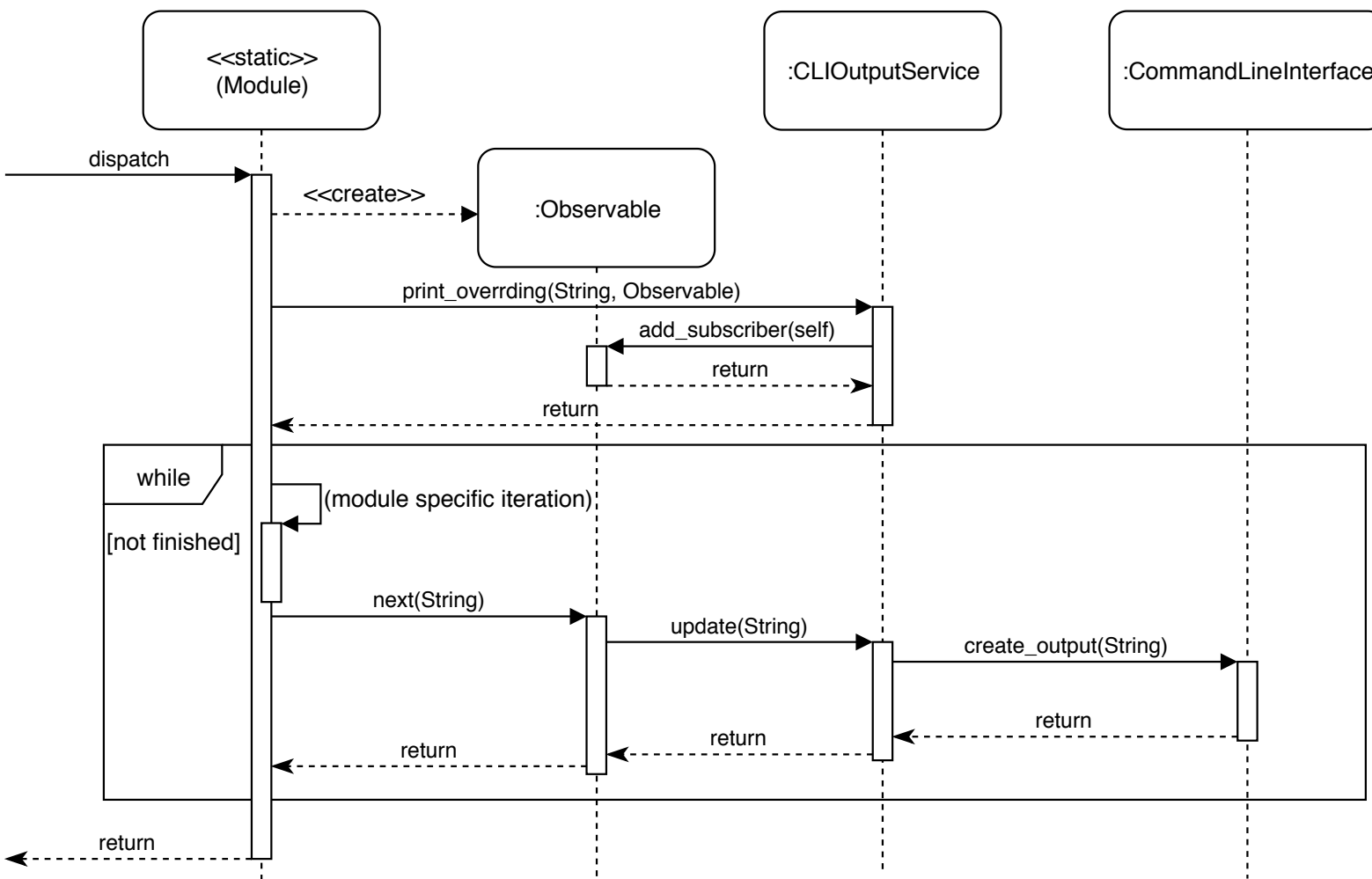
2.6 Sequence Diagrams

2.6.1 Controller Sequence Diagram



This class diagram shows what happens when the controller is being created. First it will create the `CommandLineInterface` class. After that it asks the command line interface for user input. This user input is then passed to the `CommandParser` that creates a dict out of this string and creates the right instance of a `Command`. When creating a command, it also calls the `validate` method that checks if all arguments that are needed by the module for running its computation. After that, the controller creates `CLIOutputService` and attaches the `CommandLineInterface` to it. This `CLIOutputService` is then registered to the module so this can create output to the command line interface. After all this setup steps, the module is ready to be executed. Therefore the controller calls the `execute` method on the command, which then calls the `start` method of the module with the arguments it needs. The module then starts its computations and eventually calls the `CLIOutputService` to create some user output. The module calls `print_line` with the string it wants to output. The `CLIOutputService` creates the string that can be displayed and this is passed to the `CommandLineInterface` that is attached to `CLIOutputService`. After all computations are done, the module returns his call. Now the sequence of a normal program run is finished.

2.6.2 Print Overriding String



Sometimes the user wants to print a progress like a download status that is updated by time. But there should not be a new line for each percentage the download gets further. Instead the old string should be overwritten by time and the module should only have to pass the new status. Therefore there is the method `print_` overriding in the `CLIOutputService` that wants an string and an observable object. That is why the module first creates observable object. This object is then being passed to the `print_` overriding method together with a string that will stay the same all the time. Because the `CLIOutputService` implements the `Subscriber` interface, it can subscribe itself to the observable. Now that the `CLIOutputService` is subscribed to the `Observable`, it will be notified when the `Observable` gets a new data. When the module starts its computations, it can call `next` with the new data with which it wants to override the old one in the view. In the next call of the observable, all subscribers will be notified with the new value. The string with the new value can now be printed to the command line interface.s

3 Collector

3.1 Class Description

3.1.1 Class Collector

The `Collector` class is responsible for collecting a given amount of matrices and saving it into a `HDF5` dataset. When the user types `collect` into the CLI, a collector Object will be created and the public method `collect()` with its parameters:

`amount`, `name`, `size`, `density` and `path` will be called. The class has a `Saver` class attribute and a `Generator` class attribute.

`amount`, `name`, `size`, `density` and `path` will be called. The class has a `Saver` class attribute and a `Generator` class attribute.

It uses methods from the `Generator` class to get matrices to collect and methods from the `Saver` class to save the collected dataset. (see the `collect` method Activity Diagram for a more detailed overview). The `Collector` class is the interface between matrix collecting and the CLI and conceals all the classes of the Collector described in the following.

3.1.2 Class Saver

The Saver class is just responsible for saving a given matrix dataset. Its only method is the `save(dataset, name, path)` method, which is called by the `collect` method from an Collector object.

The Saver class is just responsible for saving a given matrix dataset. Its only method is the `save(dataset, name, path)` method, which is called by the `collect` method from an Collector object.

The `save` method takes an `NPArray` as a matrix dataset, converts it into an `HDF5` file and saves it into a given directory with a given name.

3.1.3 Class Generator

The Generator class is responsible for actually generating matrices by transforming raw matrices from SuiteSparse and validating them. The `generate(size, density):Matrix` method is called by a Collector object, uses the Matrix class to initialize an empty matrix, uses the Ssget class to fetch and transform matrices from the SuiteSparse collection and uses the static `Validator.validate` method to check if the matrix is regular and can be returned.

3.1.4 Class Ssget

The Ssget class is responsible for fetching matrices from the SuiteSparse collection, transforming them and returning them. Its `getMatrix` method is called by a generator object. The `getMatrix` method uses the Matrix class to initialize a matrix, then the private `downloadMatrix` method to fetch a matrix from SuiteSparse, and after that uses its private `cutMatrix` method to cut a fixed size, regular matrix out of it.

3.1.5 Class Validator

The Validator class is a util class and responsible for validating given matrices(checking for regularity) Its only static method `validate` takes a matrix and returns true for regular, and false for not regular.

3.1.6 Class CommandLineInterface

The view represents the concrete command line interface.

The Validator class is a util class and responsible for validating given matrices (checking for regularity). Its only static method validate takes a matrix and returns true for regular, and false for not regular.

3.1.7 Class View

The view represents the command line interface.

Therefore it only consists of two methods. The first one is readInput that receives a message that will be displayed and reads the next user input. The other method is createOutput. This method prints a string to the CLI.

3.1.8 Interface OutputService

The OutputService interfaces can be implemented and passed to a module to receive the output of the modules. Therefore it has methods that represent different ways output can be displayed.

3.2 Sequence Diagrams

3.2.1 Interface Subscriber

The Subscriber interface only provides the method update() which will be triggered by an Observable upon receiving new values.

3.2.2 Class Observable

3.2.3 Interface OutputService

The OutputService interfaces can be implemented and passed to a module to receive the output of the modules. Therefore it has methods that represent different ways output can be displayed.

3.3 Interface Subscriber

The Subscriber interface only provides the method `update()` which will be triggered by an Observable upon receiving new values.

3.4 Class Observable

The Observable class can be used to notify Subscribers when new values are provided. Subscribers can subscribe themselves to an Observer to be notified get notifications. The `next()` method calls `update()` on each subscriber.

3.5 Class CLIOutputService

This class implements the `OutputService` and the `Subscriber` interface. On creation it gets a reference of the `View` to which it will pass the lines the modules wants to output. It also implements the `Subscriber` interface to subscribe itself to an observable. This can be used to display lines that are overwritten with new values like an progress bar or a counter.

3.6 Class CIIOutputService

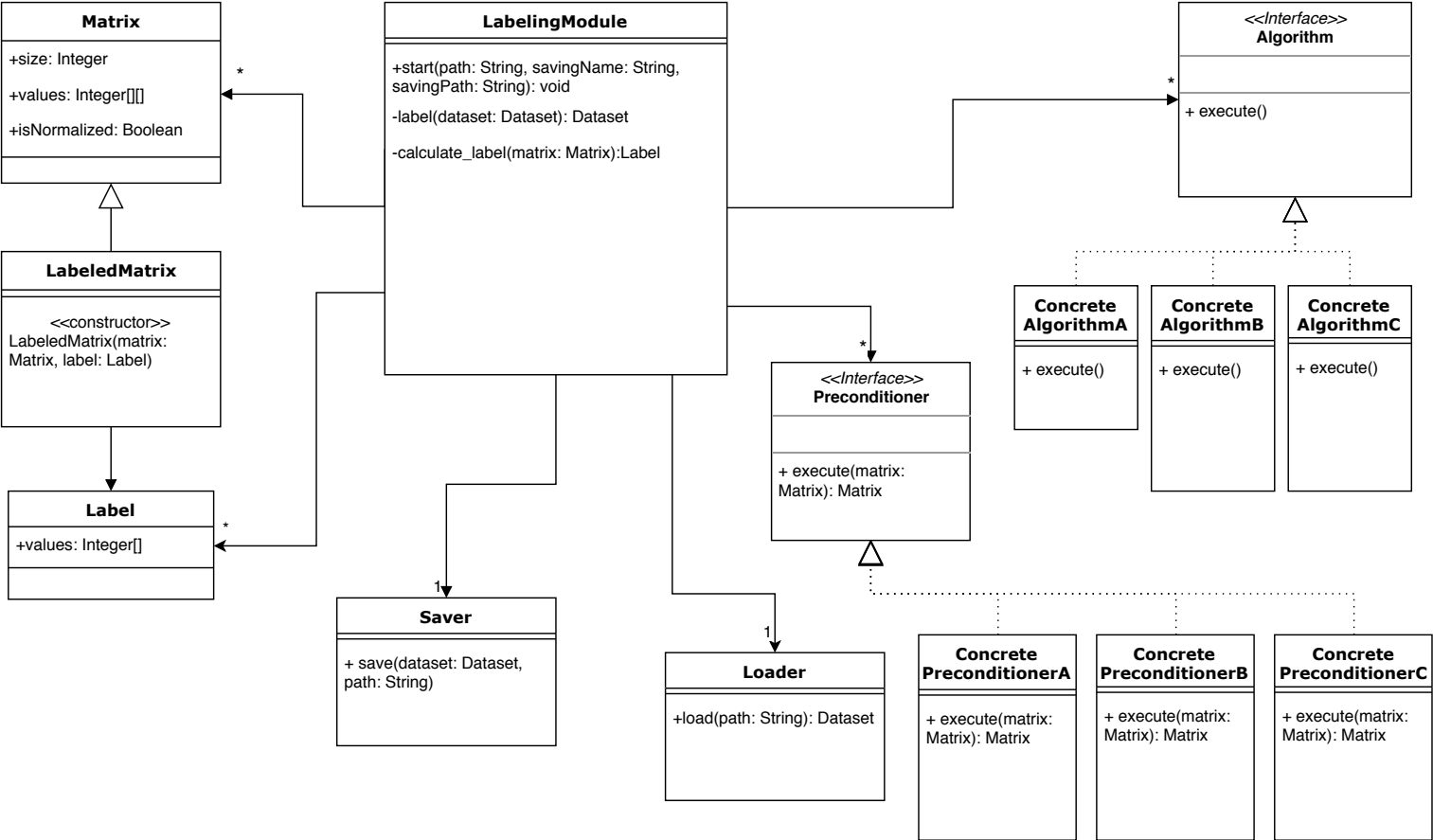
This class implements the `OutputService` and the `Subscriber` interface. On creation it gets a reference of the `View` to which it will pass the lines the modules want to output. It also implements the `Subscriber` interface to subscribe itself to an observable. This can be used to display lines that are overwritten with new values like an progress bar or counter.

3.6.1 Class CommandParser

The `CommandParser` is a class that only has one static public method. This method receives the user input string and returns a command.

4 Labeling Module

The labeling module is responsible for the labeling of the sparse matrices. The label of a matrix describes which preconditioner/iterative solver combination solves the given matrix the fastest. The label is represented by a vector. Each entry in the vector corresponds to one preconditioner/iterative solver combination. The entry of the fastest one is 1, all other entries are zero. The matrices with the corresponding labels will be used to train the neural network in the training module.



The main component of the labeling module is the class labeling module. It provides the only public method in the labling module, the method `start(path:String, savingName:String,savingPath:String)`. This method is the entry point of the module and will start the labeling process of the provided matrices(specified by the path).

The class labeling modue has a set of matrices which the module will label.

The module furthermore has a Loader class. The class labeling module has exactly one Loader class. This class is responsible for loading the matrices which get labeled. Its only method is the method `load(path:String)` which gets a path of a hdf5 file supplied and returns a dataset. If the specified path is not a hdf5 file, the programm will print an error to the command line.

Another class in the labeling module is the Saver class. The class labeling module has exactly one Saver class. This class is responsible for the saving of the matrices and the labels. Its only method is the method `save(dataset:Dataset,path:String)`. If this method is called, the specified dataset will be safed at the specified path. The matrices and the labels will be safed in one hdf5 file.

The labeling module class moreover has a validator class. This class is responsible for determining wheter the given matrix is regular. If that is not the case, the matrix will be deleted.

Since the labeling module is responsible for finding the best preconditioner/iterative solver combination for a given set of matrices, the module furthermore has a preconditioner and a Solver class. Those classes are abstract. ConcreteSolvers inherit from the class Solver and Concretepreconditioners from the class preconditioner.

The Solver class contains the logic for solving a matrix with an iterative solver. Each ConcreteSolver corresponds to one iterative solver.Each class of ConcreteSolver has the method `execute(Matrix,preconditioner)` which will solve a given matrix. We will be using the design pattern "stragety" for the iterative solvers. The reason being that each Solver does basically the same thing(solving a matrix) in a different manner. The user moreover has no influence on which Solver we will be using at any given time. Each Solver will take an optional precondtioner as its input for the method `execute(Matrix,preconditioner)`. The preconditioner will be used at every step of the iterative solver. We will be using

the design pattern "strategy" for the preconditioners too. preconditioners each return a Ginkopreconditioner which will be used by the Solver class to communicate with the ginkgo library. So each Concretepreconditioner basically does the same thing(returning a preconditioner). The user furthermore has no influence on which preconditioner we will be using at any given time. That is why the "strategy" design pattern is applicable

4.1 Class Descriptions

4.1.1 Class LabelingModule

The class LabelingModule is the main component of the labeling module. It provides one public method `start(path:String, savingName:String, savingPath:String)` which is the entry point for the labeling module. When the user types label in the command line this command will be executed. The arguments of the start method are optional. If there are no arguments provided the labeling module will be using default paths. The default paths are specified in a configuration file(see the training module for further details). If the User specifies his own paths they have to be valid. Valid in this case means that the path points to a hdf5 file with the correct formatation. If this is not the case the program will print an error message to the command line and execute a controlled shutdown so that the user can specify a correct path.

4.1.2 Class Solver

An Solver in our sense is an iterative Solver which is able to solve a linear system $Ax=b$ for x , where A is a (in our case sparse) matrix of size $n \times n$, x is a vector of size $1 \times n$ and b is vector of size $1 \times n$ ($n \in \mathbb{N}$). The matrix A in our case will be the Matrix the method `execute(matrix,preconditioner)` will receive, the vector b will be a random vector with values between 0 and 1. We chose the vector to be random since the choice of b has no great influence on the time it takes to solve the linear system. The iterative solver uses an iterative approach to solve the linear system. An iterative approach is characterized by the idea that the linear system gets solved step by step, where the solution of one step enables the solution of the next step. The class iterative solver achieves this by communicating with the ginkgo library, which has an implementation of the solvers. An iterative solver may optionally use a preconditioner for its calculation. Since there are many different iterative solvers which achieve the same outcome(solving for x) we will be using the design pattern "strategy". That is why the class `Algorithm` is abstract and `ConcreteSolvers` (which actually represent one iterative solver each) will inherit from `Solver`. The user at no point decides which Solver gets used at any given time.

A Solver has one method `execute(Matrix,preconditioner)`, which takes a matrix (and a preconditioner) and solves it. The time the iterative solver takes to solve the matrix will

be recorded and in the class `labelingModule` used to label the matrix. All `ConcreteSolvers` have to implement the abstract function `execute`.

4.1.3 Class `ConcreteSolver`

A `ConcreteSolver` is the actual representation of one iterative solver.

4.1.4 Class `preconditioner`

A preconditioner is a transformation of a linear system $Ax=b$ for x , where A is a (in our case sparse) matrix of size $n \times n$, x is a vector of size $1 \times n$ and b is vector of size $1 \times n$ ($n \in \mathbb{N}$). A transformation may be a Matrix p ($n \times n$) which would result in the linear system $PAx = Pb$. A preconditioner is used so that the linear system may be solved more easily by an iterative solver. The transformation of the preconditioner is applied in every step of an iterative solver.

The class `preconditioner` only has one method, `getpreconditioner()` which will return the `Ginkopreconditioner` corresponding to the `preconditioner` class. The `preconditioner` class achieves this by communicating with the `ginkgo` library.

4.1.5 Class `Concretepreconditioner`

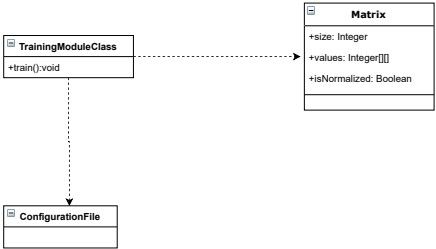
A `Concretepreconditioner` is the actual representation of one preconditioner.

4.2 Activity Diagrams

5 Training module

The training module is responsible for the training and testing of a neural network. It is structured in 2 parts, the configuration file and the class `training module`. The class `training module` loads its configuration in the configuration file. It furthermore uses a set of labeled matrices for the training and testing. With the configuration set, the class `training module` will start the training and testing. The trained network will be saved to a specified

Training Module



5.1 Class descriptions

5.1.1 Class Configuration File

The configuration file is a text file. It is used to specify all necessary information the class neural network needs to train the neural network. If the user does not change anything in the configuration file, default options will be used. The configuration file is organized in four main categories.

1. loading path of the set of matrices
2. saving path for the neural network
3. loading path for the neural network
4. model definition and hyperparameters

The loading path of the set of matrices is the path in which the matrices that are used for the training and testing are stored. The training module only supports one hdf5 file. If the path is any other file, the labling module will print an error and execute a controlled shutdown so that the user can specify a valid path. . For the training and testing making sense there should be at least 500 matrices in the hdf5 file. Otherwise the accuracy of the neural network will be so low that i can not be used for classification. If there is no path specified, the training module will use a default path. In the default path will be the latest matrices that the labling module has produced.

The saving path for the neural network is the path where the trained and tested neural network will be safed. It will be safed as a Keras model. If there is no path specified, the neural network will be safed at a default destination. If there is no path for the neural network specified in the module Classifier the module will use this default path to load its neural network.

The loading path for the neural network is strictly optional. If this path is specified the training module will use the neural network in the path for training and testing. This option enables the user to use a pre-trained neural network for training. This could be the case if the user interrupts the training process at a certain time and wants to to repeat the training later. Other use cases are of course possible too. The neural network has to be a model of the Keras framework. If the path is any other file the training module will print an error and execute a controlled shutdown so that the user can specify a valid path. If this path is not specified the training module will create a new neural

network(with the model definition and hyperparamters of the next category) and train with it.

The model definition and hyperparameters are used to determine which neural network will be trained and tested. The model definition determines the following:

- the amount of layers
- the amount of nodes in every layer
- the kind of neural network(e.g. Convolutional)
- the activation function
- the regularization
- the optimizer

The hyperparamters determine the following:

- the dropout
- the batch size
- how much of the data should be training and how much should be testing data
- the network weight initialization
- the learning rate

5.1.2 Class TrainingModule

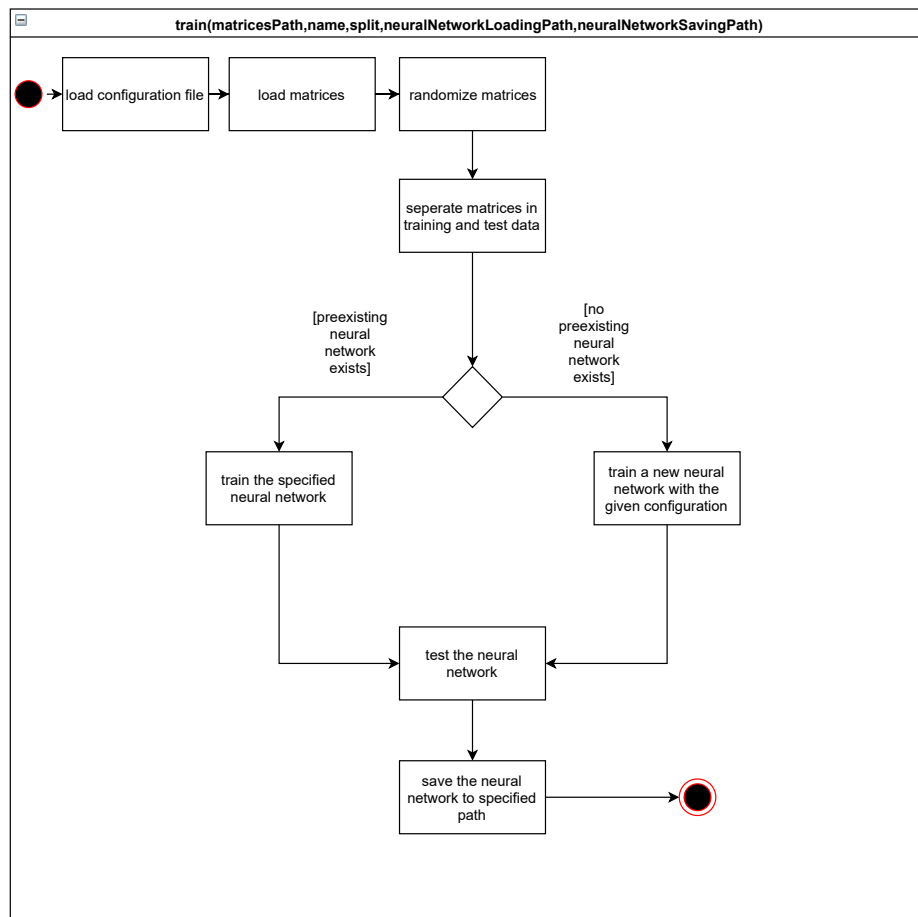
The TrainingModule class is responsible for the training and testing of a neural network. It can not be instantiated, since it is a utility class. The structure is mainly oriented towards the keras workflow and will be further described later in the activity diagramm. The class offers one public method, the method train().

We will furthermore be using the function `keras.callbacks.ModelCheckpoint` to save the neural network after every epoch. This will guarantee that we do not loose all training

progress if the computer crashes or other unexpected events happen. The procedure is consistent with the design pattern memento.

5.2 Activity Diagrams

When the user types `train()` in the CLI the method `train` in the class `TrainingModule` will be executed in the following manner:



- load the configuration file
- load the matrices
- separate matrices in training and test data
- train a preexisting neural network or a new one (depending on the configuration file)
- test the neural network
- save the neural network

The configuration file that gets loaded will be used to specify the subsequent points.

The configuration file will determine from which path the labeled matrices will be loaded. If there were no changes made in the configuration file, the default path will be used (see the class description of the configuration file). The labeled matrices will be loaded in one hdf5 file. If the path links to any other file, the class TrainingModule will print an error to the command line and execute a controlled shutdown so that the user can specify a valid path..

After that the class TrainingModule will separate the training and test data. How the data will be separated is specified in the configuration file.

Following there are two alternatives. If the user has specified a neural network in the configuration file, the class TrainingModule will train this neural network with the labeled matrices for the training. If the user has not specified a neural network in the configuration file, the class TrainingModule will create a new neural network with the specifications in the configuration file. If there are no model definitions in the configuration file the class TrainingModule will use the default neural network. The class TrainingModule then proceeds with training the new neural network with the labeled matrices for the training. In both cases the current loss will be continuously printed to the command line.

Now the neural network is trained. The class TrainingModule proceeds with testing the neural network with the labeled matrices for the testing. This process will determine the accuracy of the neural network on the given test matrices. The accuracy will be printed on the command line.

After that the neural network will be saved as a keras model. The path for the saving is specified in the configuration file.

6 Classifier

6.1 Class Classifier

6.2 Interface Solver

This interface is for the different Solvers for solving a given matrix.

6.3 Class ConcreteSolver

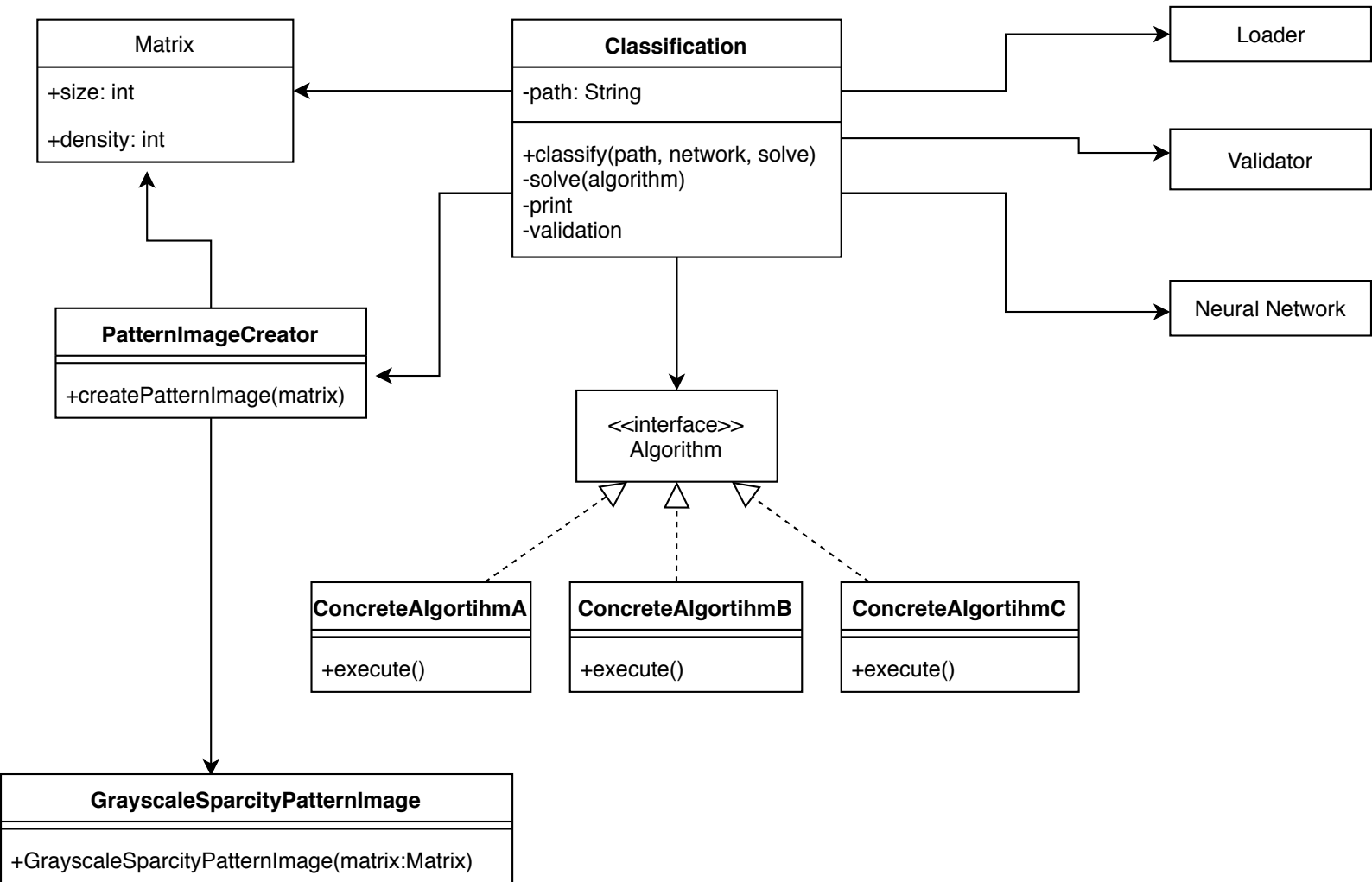
The Concrete Solver class is for solving a matrix in a certain way. This means a certain approach to solve a matrix.

6.4 Class Matrix

7 Classes which more than one Module uses

7.1 Class Loader

7.2 Class Validator



8 Glossary

Glossary

default neural network The default neural network is the network we will be using if the user does not specify a different architecture. It will be a Convolutional neural network.

iterative solver An iterative solver is a method for solving a linear system. An iterative solver uses an iterative approach to solve a matrix. An iterative approach is characterized by the idea that the matrix gets solved step by step, where the solution of one step enables the solution of the next step. An iterative solver may use a preconditioner to improve its results..

memento Memento is a design pattern which provides the ability to restore an object to its previous state. In our case the object is the neural network. After a certain amount of training steps we will save the state of the neural network. If the program crashes at a certain point in time we will be able to reset the state of a neural network to a safed state. That way we will not loose much of our training achievements..

preconditioner A preconditioner is a Transformation of a linear System. With certain preconditioners, certain iterative solvers may solve a linear system faster that with no preconditioner. The transformation of the preconditioner gets applied in each step of the iterative solver..

strategy strategy is a design pattern of the category behavioral design patterns. It defines a family of iterchangable algorithms. It furthermore enables the selection of an algorithm at runtime, so that the algorithm may vary independently of the clients that use them..