Genome Sequencing Program

Geert Haerens

May 2023

# Requirements

Original specification from Prof xxxxx of the KUL :



## Requirements analysis

**The following basis data types are identified:**

* R1: A Genome Sequence, consisting of an identifier and a genome sequence
  + It is further specified that a Genome Sequence can be an genome sequence that is part an Alignment and on which a Bioinformaticians is performing actions
  + Or is a reference genome sequence, used in an Alignment to perform certain operations against.
* R2: An Alignment, consisting of a multiple genome sequences
  + It is further specified that an Alignment can be an Alignment that is assigned to a Bioinformaticians.
  + Or can be considered as the Optimal Alignment
* R3: A user, consisting of a first name, fast name, age and role.
* R4: A role, being a Team Leader, Bioinformatician and Technical Support
* R5: A user list, consisting of users.

The requirements talk about a repository, but it is not clearly specified what is meant by it. I have taken the assumption that there must be a data type for the Repository that contains the list of users, the Optimum Alignment and all the Alignments on which Bioinformaticians are working.

**The following basic operation are identified:**

* R6: Creation of the “Standard Alignment”, of an Alignment
* R7: Creation of the “SNiP Alignment” of an Alignment
* R8: Creation of the “Alignment Score” of an Alignment
* Manipulation of Genome Sequences
  + R9: Search for a Genome Sequence based on the id, in an Alignment.
  + R10: Search for Genome Sequences that contain a certain sub genome sequence, in an Alignment.
  + R11: Replace a sub genome sequence with another sub genome sequence of equal length, in a genome sequence.
  + R12: place a sub genome sequence with another sub genome sequence of equal length, in all genome sequences of an Alignment.
  + R13: Add a Genome Sequence to an Alignment.
  + R14: Remove a Genome Sequence from an Alignment.
  + R15: Replace a Genome Sequence of an Alignment with a new Genome Sequence.

**Depending on the role of the user, different extra operations are possible:**

*As Team Leader:*

* R16: see the “Standard Alignment”, “Snip Alignment” and “Alignment Score” of the Optimal Alignment
* R17: see the “Standard Alignment”, “Snip Alignment” and “Alignment Score” of all Alignments assigned to Bioinformatician
* R18: assign a copy of the current Optimal Alignment to a Bioinformatician so he/she can manipulate the Alignment.
* R19: Promote an of the Alignments of a Bioinformatician to the Optimal Alignment

*As Bioinformatician:*

* R20: Perform the basic Genome operation R6 to R8
* R21: Perform the basic Genome manipulations R9 to R15
* R22: Write the assigned Alignments to a file.
* R23: Write the “Alignments Scores” of the different Alignments to a file.
* R24: See/load the Assigned Alignments

*As Technical Support:*

* R25: Backup the Repository
* R26: Restore the Repository
* R27: Clearing the Repository

**Additional Information**

A .fasta file contains an Alignment on which everybody will work.

A team.txt file contains the users of the applications.

All users specified in the team.txt file should be able to perform their operations.

# Architecture and Design

Based on the requirements I had an idea about what the application should look like. There were still some topics related to the repository and the role that were not clear, but one thing was for sure, I needed data types to get started.

I plan to structure my program in accordance with NS principles and the SOLID principles. I create a package called BeBl – Business Entities and Logic – in which I want to put all the classes related to the repository and business logic.

A second package will contain the Use Cases, basically from R9 to R27.

I also had the idea to use a TDD approach. The Junit package was integrated in my JAVA project and for each class I will create in the BeBl or UC package, I plan to first identity the test with the

GIVEN

WHEN

THEN

RESULT

structure , and consider the implementation of the class as completed if all tests work.

I decided to apply a naming convention to all my classes. All data classes start with a D and all task classes start with a T. After the D or T, there is an underscore and the abbreviation of SOLID architecture layer to witch the class belongs (BeBl, UC, CCGP (checkers, converters, gateways, presenters), UI and CCC (cross cutting concerns)).

With this basis structure in place, I started programming.

# Programming the Requirements

## Implementing R1:

Genome Sequence is made up of 2 strings. One for the ID and one for the actual sequence. The sequence should only contain the characters A, G, T, C but I decided not to implement a check of the validity of the characters.

The class **D\_BeBl\_GenomeSequence** contains the getters and setters for the 2 strings, and one getOverview method. In the specifications it is indicated that the visualization of the genome id happens by first showing a “>” followed with the actual id. This convention is integrated in the getOverview.

The constructure of the class create an object with ID = “empty”. The correct ID of the Genome Sequence object needs to be set with setters.

Unit test **Test\_D\_BeBl\_GenomeSequence** was created to test the methods.

## Requirements R3 and R4

As Alignments will be assigned to users, I first need to make the necessary classes for handle users before starting to create lasses to handle Alignments.

The requirements state that a user has a first name, last name, age and role. As I did not see the added value of age, I decided to ignore it and have it not be part of the **D\_BeBl\_User** class. The class contains 2 strings – first name and last name – and a role that is part of an enumeration.

I opted for an Enumeration type so I could define the roles better. However, I do not fully understand the benefits and methods associated with the Enumeration type.

**Learning:** I don’t further exploit the benefits of the enumeration type and have doubts if is was a good implementation choice.

The possible user roles are: “NONE”, “TEAMLEAD”, “TECHNICALSUPPORT”, “BIOINFORMATICIAN”.

The initial values of role is “NONE”, “INVALID\_USER” for first name and “INVALID\_USER” for last name.

The **D\_BeBl\_User** class contains setters and getters for role, first name and last name and a getOverview method.

A unit test was created to test the different methods**: Test\_D\_BeBl\_User**.

## Implementing R5

The **D\_BeBl\_UserList** class contains an ArrayList of **D\_BeBl\_User** and has a name.

There should be a setter for both but there is only and addUser method in the class. The class also contains a getUserID method which returns the index of that user in the ArrayList of users and a getUser method which returns the user for a given UserID. These methods were created to make user manipulations a bit easier or better, quicker to program. Manipulations of users are not specified in the requirements but it seems a valuable addition to the program.

There also is a removeUser method. All the logic of this method is not yet implemented as when a user gets remove, all the Alignments assigned to him should be either removed, or re-assigned, a something else… The specification don’t say what to do in such case. Ideally, the creator of the specifications should be contacted for more guidance.

The class contains a getOverview, which gives the users and the ID of the user (index in the ArrayList).

A unit test was created to test the different methods: Test\_D\_UserList.

## Implementing R2:

An Alignment is made up of a name (String), an owner (a user – see R3, R4, R5) a genome sequence that serves as the reference genome sequence, and an ArrayList of genome sequences.

Name, owner and ref genome sequence are not explicitly added in the requirements. However, to distinguish different Alignments in the Repository, I deemed it necessary to give each Alignment a name. As Alignments are to be assigned by the Team Leader to the Bioinformaticians and as the Bioinformaticians must know the Alignments assigned to them, I decided that each Alignment required an owner which is set by the Team Leader and used to search for all the Alignments assigned to a Bioinformatician.

The requirements talk about initializing the repository and emptying all Alignments. To recognize an empty Alignment easily, the default name of the Alignment at object creation is “empty”.

The class **D\_BeBl\_Alignment** contains methods to set the name, reference genome sequence, the owner and to add genome sequences (called sample sequences). The implicit order for proper initialization and setting of the **D\_BeBl\_Alignment** should be:

* New **D\_BeBl\_Alignment** objects 🡪 get the name “empty”
* Set the name
* Set the owner
* Add sample genome sequences
* Set the reference genome sequence if required

The first added sample genome sequence becomes the reference genome sequence (by convention and mentioned in the requirements document). If required, this can be changed with the setReferenceGenonmeSequence method.

The class also contains a getOverview method that visualizes the Alignment – name, owner, ref sequence, all sequences.

No Unit test was created to test the different methods because entering all the data of an alignment in a test would take too much time (there is an impty unit test called **Test\_D\_GenomeSequence**). What I needed at this point was the possibility to load the provided HIV.Fasta file in an Alignment object.

## Loading the .Fasta file

The HIV.Fasta file, provided with the specifications, is an Alignment. As the different manipulations needs to demonstrated on this file, it needs to be loaded. At this point it makes sense to create the necessary classed for loading of this file at they can then be used to create tests on manipulations on the Alignment.

As we read from file, but want to potentially be able to read from something other data sources, I decided that reading from a file must be a class that is part of the Cross Cutting Concerns.

A classe **T\_CCC\_FileRead** is created that reads the content of a given file (full path) into a String. With this class, we can read the content of the HIV.Fasta file. The resulting String now needs to be converted into an Alignment object.

For this the **T\_UC\_AlignmentLoader** is created. The task takes the full name of the .fasta file as input.

For both the **T\_CCC\_FileReaed** and **T\_UC\_AlignmentLoader**, unit tests were created – Test\_T\_FileRead and Test\_T\_AlignmentLoader (note the violation of my own naming convention).

**Learning:** This was not a good implementation as it should be that the AlignmentLoader hides from its calles, the technology used to get the data – in this case a file. But by having to give a filename as variable, you achieve just the opposite. It should rather be that one gives the alignment name, like HIV and that there is a convention enforced that that file that contains the HIV alignment is called HIV.fasta. This logic would be part of the AlignmentLoader.

## Loading the team.txt file

The **T\_CCC\_FileReader** is also used to read the provided team.txt file. The FileReader provides a String with the content of this file. This String is converted into a UserList object by means of the **T\_UC\_UserListLoader.**

**Learning:** Same remark as for I1, the UserListLoader takes a file name as input, while is should get a filename agnostic name for the user list as input. The UserListLoader could contain the logic to construct the proper filename and use the **T\_CCC\_FileReader** for loading the content.

A unit test called **Test\_T\_UserListLoader** (note the violation of my own naming convention) was created

## Loading the Repository

With the ability to load the .fast file and the team.txt file, I decided to create the repository class **D\_BeBl\_Repository.** It contains a name, an Alignment that is registered as referred to as the Optimal Alignment, an ArrayList of all Alignments on which Bioinformaticians are working (the workAlignments) , and the UserList.

The class contains setters and getters for the name, Optimal Alignments and UserList. It also contains a getInitialOverview method to show the default loaded repository.

**Learning:** The **D\_BeBl\_Repository** should not have contained the getIntialOverview method it only provides the overview for the initial load and cannot be used the show the full repository content once the users start making changes in it.

As the only input/test data I have is the HIV.fast and team.txt file, I decided to hardcode the names of those 2 default files, together with the default repository name and the default location of the repository. I wanted to keep the option open of changing those by adding setters and getters for the fast and team file location, and their names.

**Learning:** This decision made my life earlier for loading the test data but is clearly a bad idea from the point of view of using different .fast files and different team files in the future. I realized that later and tried to find another solution (to be continued).

I knew that the solution has to work with different files then the HIV.fast and team.txt file. Yet at this point I kind of ignored it and continued with the creation of the **T\_UC\_RepositoryInitiation**, to be used for loading the HIV.fast and team.txt file, instead of making **a T\_UC\_Repositoryloader,** usable for any kind of fast file and team file. The class **T\_UC\_Repositoryloader** exists but is empty (see later).

The **T\_UC\_RepositoryInitiation** creates a Repository, with default name = HIV, from the default fasta and team file. The HIV Alignment is the “Optimum Alignment” and the workAlignments ArrayList is empty (nothing assigned yet to Bioinformaticians).

A unit test called **Test\_T\_UC\_RepositoryInitiation** (aha, I again follow my own naming conventions) has been created.

## Implementing R6

The “Standard Alignment” is a visualization of the different genome sequences that are part of the alignment. The different genome sequence strings and their ID as shown on the screen.

I consider the creation of the Standard Alignment as Business Logic. The task to create the Standard Alignment is part of the BeBl package – **T\_BeBl\_StandardAlignment**.

The task creates a String that contains the visualization of the Standard Alignment. The string looks a lot like the result of the getOverview of the **D\_BeBl\_Alignment**. It does begin with a banner stating that this is the “Standard Alignment”, while the getOverview is not.

**Learning:** At this point it’s clear that I introduce redundant code and multiple variations. This was not a good idea and should be solved.

A unit test was created to test the task: **Test\_T\_BeBl\_StandardAlignment.**

## Implementing R7

The “SNiP Alignment” shows the differences between the reference gnome and all other genomes. I also consider this task as Business Logic and part of the BeBl package – T**\_BeBl\_SNiPAlignment**. Each genome sequence of the Alignment is compared to the Reference Genome Sequence and the differences are shown. The original specifications contain an example.

A Unit test was created to test the task: **Test\_T\_BeBl\_SNiPAlignment.**

## Implementing R8

The Alignment Score counts the number of differences in the different Genome Sequences and the Reference Genome Sequence. It is a procedure similar to the SNiP Alignment but instead of adding the difference to a visualization string, a counter is increased. This task is also considered as Business Logic **T\_BeBl\_AlignmentScoreCalculator**.

A Unit test was created to test the task: **Test\_T\_BeBl\_AlignmentScoreCalculator.**

**Learning:** Task are the programming of verbs. While this is clearly reflected in the name of the **T\_BeBl\_AlignmentScoreCalculator**, it is not in the **T\_BeBl\_StandarAlignment** and **T\_BeBl\_SNiPAlignment** tasks. For anthropomorphic reasons, their name should be **T\_BeBl\_StandarAlignmentCreator** and **T\_BeBl\_SNiPAlignmentCreator.**

## Implementing R9

The class **T\_UC\_GenomeInAlignmentSearcher** willlook in an Alignment with a Genome Sequence with a provided ID. As soon as a march is found, the search stops. This means that I make an implicit assumption that genome sequence IDs are unique in an Alignment (not part of the specifications).

A unit test exists to test the searcher: **Test\_T\_UC\_GenomeInAlignmentSearcher.**

## Implementing R10

R10 has been implemented twice. Once as **T\_UC\_SequenceInAlignmentSearcher** and once as T\_UC\_SubSequenceInAlignmentSearcher.

The **class T\_UC\_SubSequenceInAlignmentSearcher** will look for Genome Sequences that contain a specified subsequence (AATTGC for instance). The class returns an ArrayList of GenomeSequences.

The **class T\_UC\_SequenceInAlignmentSearcher** will look for Genome Sequences that contain a specified subsequence (AATTGC for instance). The class returns an ArrayList of Strings, where the Strings are the GenomeSequence IDs.

**T\_UC\_SequenceInAlignmentSearcher** has a test: **Test\_T\_UC\_SequenceInAlignmentSearcher** but is not used in the program.

**T\_UC\_SubSequenceInAlignmentSearcher** has no test, but is used in the program.

**Learning:** It is for me unclear why I created the same functionality twice but with different return types.

## Implementing R11

**T\_UC\_ReplaceSequenceInGenomeSequence** replaces a subsequence of a genome sequence by a subsequence of the same length.

A unit test **Test\_T\_UC\_ReplaceSequenceInGenomeSequence** has been made.

## Implementing R12

**T\_UC\_ReplaceSequenceInAlignment** replaces a subsequence in all genome sequences that are part of an Alignment.

A Unit **Test\_T\_UC\_ReplaceSequenceInAlignment** has been made.

## Implementing R13

Adding a Genome Sequence to an Alignment is in itself a simple task to do. However, the provided specifications indicate that in the example HIV.fast file, each Genome Sequence consist of 2500 nucleotides (or characters). It seems not very practical to manually input such a sequence. As this touches the input/output of the program, I decided to wait with the implementation of this topic until I start working on the UI. I basically postponed the implementation because I doubted how to handle it.

## Implementing R14

Removing a Genome Sequence from an Alignment requires first to check for the existence of a Genome Sequence in an Alignment and if it exists, to remove it. For this I decided to develop the **T\_UC\_GenomeInAlignmentSearcher**. The task takes the Genome ID as input and looks for the Genome Sequence with that ID in the Alignment. The task returns the Genome Sequence object that is in the Alignment. If it does not exists, it returns an empty Genome Sequence object. Recall from the implementation of R1 that a default initiated Genome Sequence object has as ID “empty”. If the search does not find the correct Genome Sequence in the Alignment, you get such a default initiated object as return. Checking the ID string tells you if the search was successful or not.

A unit test **Test\_UC\_GenomeInAlignmentSearcher** has been made.

**Learning:** The handling of a failed search is not a very elegant implementation. I think there is room for improvement.

The remove requirement is realized in **T\_UC\_RemoveGenomeFromAlignment.** There are some special cases to be taken into account such as, what to do when there is only one Genome Sequence in the Alignment (no guidance found in the provided specifications) and what if the to be remove Genome Sequence is the Reference Genome Sequence of the Alignment. Although the code takes those cases into account, the unit test **Test\_T\_UC\_GenomeInAlignmentSearcher** does not test these cases!

**Learning:** Unit test must test all the cases and specially the exceptions. Looking at the code of the **T\_UC\_RemoveGenomeFromAlignment**, I even think that there is an error in the handling of the different cases!

## Implementation R15

Replacing a Genome Sequence in an Alignment confronted me again with the problem of “how to enter a 2500 char sequence for a new Genome Sequence”. The provided specification give no additional guidance on the replacement. Is a replacement a simple remove and add or does the add need to be done at the same location in the list of Genome Sequences as where the remove was performed? I assumed the latter.

When I wrote the code for **T\_UC\_ReplaceGenomeInAlignment**, I was in a more inspirational mode compared to when I avoided writing R13. I used the existing task to create a new Genome Sequence, by taking an first an existing Genome Sequence, giving it a new ID and replacing a subsequence. I then had the required test data to write the unit test **Test\_T\_UC\_ReplaceGenomeInAlignment**.

**Learning:** While looking at the code of **T\_UC\_ReplaceGenomeInAlignment,** I notice that this task and the other remove/replace tasks, do not return something. They do provide an error message when something is not OK, but the code calling this object’s method, is unaware of the state. This is not a good idea as it violates SoS, while I had the intension of creating code that take NS principles into account.

## Interaction with the users

At his point, a lot of the required manipulation on Alignments and Genome Sequences are created. The question now becomes, how will they be called. From R16 to R27 we know that different roles can perform different tasks. This means that we require a kind of login system where the users identifies himself with his first name and last name that the his role needs to be determined. Based on his role, different tasks are available.

Doing tasks without data is useless. The program needs to load the .fast and teams file into the repository before logon can happen.

The provided specification say nothing on how to treat the multi-user dimension of the application. As this exercise is for students who have had a basis JAVA programming course, I assume that an application that runs in multiple threads for different users, is not the kind of expected solution. A simpler form of implementation is more realistic.

The provided specifications state that no UI needs to be created. The program must simply run from the console. The console is of course also a form a UI, simpler, but input and output will need to happen via the Console.

Input: logon information, choice of action, action relevant data.

Output: result of the actions, textual information about the available operations.

**Based on this, a list of extra requirements is created.**

**R30:** Load a repository at startup of the program. This can be the default repository (hiv.fast and team.txt), a new repository (different fast file and team file), or an existing (previously created repository.

**R31:** R30 must provide as a choice, including the possibility to quite the program.

**R32:** A user must be able to logon and logoff of the application. Only one user can work in the repository at once. Like in the good-old-times when 20 people were sharing 1 PC 😊.

**R33:** After loading a Repository, a user must logon by providing his first and last name. Based on the role of the user, the users will get a menu with all possible and allowed activities.

**R34:** Each user must be able to perform a logoff, allowing somebody else to start working on the repository.

## Implementing R30 and R31

The interactions with the user are done via the console UI. According to SOLID, the code belongs for UI belongs to the UI layer which is positions on the outer side of the clean architecture model. Dependencies should point inwards such that this code depends on layers below, and not the inverse. Practically this means that all code created up till know in the BeBl, UC, CCC layer, should not be impacted by what will be programmed in the UI layer.

R30 is realized by **M\_UI\_Startup** and the **Main** class. When Main is executed, the Startup object provided the menu regarding repository loading.

* Load the default repository - hiv.fast and team.txt embedded in the Repository object
* Custom initial repository load – create a new repository with a new .fast and team file
* Load an existing repository – load a repository that contains work of different bioinformaticians.
* Quite the program

The read from the console happens via the Scanner class.

The user needs to enter his choice by means of a number corresponding to the required action.

A switch statement will launch the appropriate action based on what is read by Scanner. The default action lets the users re-enter his input as it does not correspond to the allowed actions. The Startup menu is being looped till a valid input is being provided.

When creating a JUNIT test for **M\_UI\_Startup**, I got an error message saying that JUNIT test are supposed to run in an automated fashion and that input from console is not allowed. As I did not know how to solve this immediately, the only option is saw what to start creating the Main class and test the program from there.

## Implementing R32-R33-R34

When a repository has been loaded, the logon of the user can happen. The Main program launches a **M\_UI\_Logon** object.

The object ask for the first name and the last name of the user via Scanner. The link must now be made between the entered user info and the defined users in the Repository (Loaded via the team.txt file). For this the **T\_UC\_UserSearcher** is created. It takes the first and last name as input and return a the corresponding D\_BeBl\_User object. If such an object does not exists, it returns an empty **D\_BeBl\_User** object. Recall that an empty has as first and last name “INVALID\_USER”.

A unit test for **T\_UC\_UserSearcher** was not created as I was kind of entering the no-more-JUNIT-tests-phase here.

The **M\_UI\_Logon** loops till either a valid user has been provided or the choice to quite the logon is provided. **M\_UI\_Logon** returns the active user to **Main**.

The Main class takes the active users and extracts the role from it. A Swich statement is used to launch the appropriate task screen (for teamlead, technical support of bioinformatician).

After execution of the tasks corresponding with a role, a new user can logon to the repository or quite the program.

**Learning:** A lot of things are going wrong with the implementation of R30-R32. The creation of unit tests was abandoned as I was to eager to deliver functionality instead of taking a step back and questioning how I could test console input. The root cause is that I started violating the SOLID Clean Architecture. The M\_UI classes should have only contained the showing of the option menu and reading of the input and given it to classes from the CCGP layer (checkers/converters, controllers, gateways, presenters). Those should have contained the Switch statement based in the input from the scanner, handled incorrect input or launch the action corresponding with the input.

Once I started on this path, I continued it … I went to the dark side.

My Main class contains code to handle the logon. The **M\_UI\_Startup** uses the UC layer and uses objects from the BeBl Layer, while this all should have been hidden from it. But at that point, I wanted to fast. I had all the important classes lined up and wanted to see a working program asap.

The Main class contains 2 Do loops with too many implementation details. Far too much code to reside in Main. It should have been in the M\_UI\_xxx classes.

## Implementing the Team Leader

According to the provided specifications, a Team Leader must be able to:

* Show the Optimal Alignment’s Standard and SNiP Alignment and the AlignmentScore.
* View all assigned Alignments (work assigned to Bioinformaticians).
* Assign a copy of the current Optimal Alignment to a Bioinformatician.
* Promote an Alignment of a Bioinformatician to Optimal Alignment.
* Write all Alignments of the Bioinformaticians to file.
* Write all Alignment Reports to a file.
* Quite working as Team Leader.

The above choices are presented via the **M\_UI\_TeamLeaderOperations** class. As is the case with the M\_UI\_Startup, the class violates the Clean Architecture as it uses classes from the UC and BeBl layer directly. In class, a mix of taking in put and processing is also combined. The implementation of handling of the different use cases is writing in this class, basically having contain business logic into the UI.

Showing the Optimal Alignment in Standard Alignment and SNiP Alignment format, and the Alignment Scope is just matter of calling the associated tasks. However, I notice that some of the information that will be shown on the console is part of the **M\_UI\_TeamLeaderOperations** class, while other is part of the **T\_BeBl\_StandardAlignment** and **T\_BeBl\_SNiPAlignment** classes. This would mean that a change in UI tech could require changes to these tasks!

**Learning:** When creating the convenience method for getOverview, one should stick to the essentials and not worry about UI experience.

Showing the Assigned Alignments is just a matter of showing the Alignment name, owner (person to whom it is assigned) and the score, for alle Alignments in the Repository. This should have been a task in the UC layer, but due to already mentioned reasons, this piece code is part of the **M\_UI\_TeamLeaderOperations** class.

The assign an Alignment to a Bioinformatician, I first show the list of users and ask to enter the ID of the user who needs to get the Alignment assigned. If a wrong or invalid ID is provided, the input needs to be re-entered. I notice now that you can’t get out of this loop if you what to quite try to assign an Alignment :-/.

**Learning:** always foresee a way to escape an input loop.

The Bioinformatician gets a copy of the Optimal Alignment, not a reference to the Optimal Alignment, as he/she needs to be able to perform operations on the Alignment without modifying the Optimal Alignment. The copy is performed via the **T\_UC\_CopyAlignment** class.

We must be able to distinguish the different assigned Alignments from each other. We need them to give names that are different than “Optimal Alignment”. It seems logical to include the name of the Bioinformatician in this name, but we also require a version/time differentiator as one Optimal Alignment can be assigned multiple times to a Bioinformatician. I opted for a concatenation of first name, last name, and current system time in ml, all delimited for each other via a “\*”. This piece of logic should have been put in a separate class. And all the assignment logic should have been put in a separate class as well (see my previous sin).

Promoting one of the assigned Alignment to the Optimal Alignment required entering the name of the Optimal Alignment, searching the object corresponding with that name in the Repository, and putting a copy of that Alignment object int the Optimal Alignment of the Repository. For the search, a new class **T\_UC\_AlignmentNameSearcher** was created.

In the best of catholic traditions, I have nothing but sinned in the implementation of this part of code, sinned against the laws of modularity.

**Learning:** At this point, I see that in my code I am not consistent in the way I send data to different use cases. Sometimes I provide as parameters the bare minimum – such as the string that is the AlignmentName – and sometimes I just give the whole Alignment and let the extraction of the AlignmentName to be handled by the called method. I apply data coupling in the former and stamp coupling in the latter. From an evolvability point of view stamp coupling is preferred. I must work with this in a more consistent way.

Because I haven’t figured out yet how I want to persist the Repository, I decided to wait for the implementation of the write assigned Alignments and Alignment report to file.

## Implementing the Technical Support

The provided specifications indicate that Technical Support must be able to do the following:

* Backup the Repository.
* Restore the Repository from a Backup.
* Initiate the Repository with the default HIV.fast and team.txt file.
* Initiate the Repository with a new .fast and team file.

I decided to add the addition and removal of users to this list as well.

These task are made available with the **M\_UI\_TechnicalSupportOperations** class.

Because I haven’t figured out yet how I want to persist the Repository, I decided to wait for the implementation of the Backup and Restore operations.

Initiation of the Repository should be re-initiation as at program start, an initial initiation is done. The re-initiation to the default uses the existing task **T\_UC\_RepositoryInitiation**.

**Learning:** I notice that I sometimes have task to return a result and sometimes I have task that return nothing but have manipulated the provided variables. Although both are valid (I think), it does not help the readability of my code. I must figure out a more consistent approach.

The re-initiation with a new fast file and team file, uses the **T\_UC\_RepositoryInitiation** as well, but starts then by replacing the default .fast file by the new provided fast file and idem for team file. Before **T\_UC\_RepositoryInitiation** can be used,condole input for the file names and their location must be provided.

**Learning:** At this point, the decision to include in the **T\_UC\_RepositoryInitiation** the HIV.fast and team.txt file and their default path, seems like a bad idea. I should have separated these concerns and made a more generic repository initiator.

At this point in time, I really started to see that I massively violated the principles I preach. I decided to try to do better for the implementation for Add and Remove Users. The idea should be that the only thing that the UI does is show some the user information about the task, asking and getting the input and showing relevant output. The **M\_UI\_TechnicalSupport** class prints some basic information and directions to the user and then calls the checking **T\_CCPG\_AddUserChecker.** In this checking, the user is requested to enter the first name, last name and role. The checker grabs the input from the console. Whatever is entered as first and last name is taken without further checking. The role is checked – must be one of the 3 valid roles. If not, the information needs to re-entered.

**Learning:** Although my intensions were good, there still are some issues with this implementation. First of all, the checking does not only check, but he also adds the user to the Repository. In that respect it is first a controller, and that controller should have called a checker. From the UI, only the controller should be visible. In the **T\_CCPG\_AddUserChecker** information printed to the console. This should also be avoided as it means that the class depends on the UI console**.** The UI should have taken care of grabbing the input from the console UI and just passing it to the Checker/Controller for further validation and execution. I also noted that as long as no valid role is entered, the are stuck in a loop. An escape should be provided.

For the removal of a user, the same approach as for adding a user was used. The **M\_UI\_TechnicalSupport** prints info to the console and then called the T\_CCPP\_RemoveUserChecker to perform the action. The implementation has from the same shortcomings as mentioned in previous learning.

The original specification do not tell us what to do with assigned Alignments when a user is removed from the repository. I have not included additional logic to handle such cases.

## Implementing the Bioinformatcian

* The Bioinformaticain can perform the following tasks:
* List the Alignments assigned to him.
* Lis the AlignmentScores of all the Alignments assigned to him.
* Search through all the sequences in the Alignments for a sub sequence.
* Replace a sub sequence in all genome sequences of an Alignment.
* Replace a sub sequence in one genome sequence in an Alignment.
* Show the Standard Alignment of an Alignment.
* Show the SNiP Alignment of an Alignment.
* Remove a genome sequence from an Alignment.
* Replace a genome sequence with another genome sequence.
* Write an Alignment to file.
* Write an Alignment Report file.

These tasks are available via the **M\_UI\_Bioinformatcian** class. The good intentions I expressed while implementing the add and remove user tasks, were forgotten while implementing the **M\_UI\_Bioinformatcian** class. The result is a class of 319 code lines and not all functionalities are implemented yet.

All code-craftings, good and bad, previously mentioned and be found in the coding of this class as well.

A few extra classes were created such as:

**T\_UC\_AssignedAlignmentSearcher**: to look for the Alignments assigned to the Bioinformatician.

**T\_UC\_AlignmentNameSearcher**: to get the Alignments on which the Bioinformatician will perform tasks.

**T\_UC\_SaveAlignment**: to save an Alignment.

**T\_UC\_SaveReport**: to save a Report.

**T\_CCC\_FileWrite:** save the file (either the Alignment or the report, or any other string).

When implementing the Save tasks, I came across the limitations of my previous design choices for the Repository like:

* Where will the Repository be saved?
* What will a saved-Repository look like on the file system?

The **D\_BeBl\_Repository** class only contains a path to the .fast and team file and this is insufficient

The save operations should solve the open questions.

I decided to create a special class called **D\_CCPG\_Environment** that contains:

* The folder containing the program executable
* The folder containing program data (like new .fast and team files)
* The folder containing the Repositories (like HIV)
* The folder containing the Backups of the Repositories.

In a Repository Folder, the different saved assigned Alignments, reports on Alignments, the optimum Alignment and the users should be saved to file. Recall that I did not have implemented the **T\_UC\_LoadRepository**. With the above information I could create the class it would read the mentioned files with a repository location.

Working with the **D\_CCPG\_Environment** class in combination with a naming convention for file names (based on the unique name of the objects), eliminates the need to pass a filename to the **T\_UC\_SaveAlignment** and **T\_UC\_SaveReport.** The Save/write tasks are such separating the concerns of deciding what to save and actual persistence in technology

**Learning:** the D\_CCPG\_Environment data is currently assigned to the wrong SOLID layer. The UCs depend on it while UCs should only depend on classes from the BeBl layer. To be decided where this is best located and adjust the code accordingly.

# Appendix – Package Structure

A screenshot of a computer

Description automatically generated

# Appendix – source code on GitHub

\Assignment 🡪 the original assignment

\src 🡪 the java source files

\Data 🡪 the data structure during program execution 🡪 see the D\_CCPG\_Environment class

The D\_CCPG\_Environment class needs to adjusted to your work environment