PROGRAMMING LIFE CONTEXT
DELFT UNIVERSITY OF TECHNOLOGY

DNA is Not an Acronym

Final Report

Joël Abrahams - joelabrahams - 4443268
Georgios Andreadis - gandreadis - 4462254
Casper Boone - cboone - 4482107
Niels de Bruin - ndebruin - 4375440
Felix Dekker - fwdekker - 4461002

June 2017

Contents

1 Introduction

Our life is governed by a string of 3 billion tiny units (NHGRI, 2003). While there have been incredible advances in the past years towards fully understanding our own 'source code', a lot still remains to be discovered. These discoveries have the potential for high impact in our society: Breakthroughs in the development of treatments for a variety of diseases would benefit us all. Valuable insights lie in the differences between a large number of genomes of the same kind of organism. What parts remain largely the same? Where are many mutations? The scientific community needs tools to visualize and work with large numbers of genetic source codes efficiently and intuitively, in order to be able to derive useful insights for its research.

To this end, we present `Hygene`, a multiple genome graph visualizer. It enables scientists to analyze and compare multiple versions of the same genetic material in a graph format. `Hygene` allows researchers to navigate through such graphs intuitively, as well as easily identify mutations between different genomes. Additionally, users can place bookmarks at interesting locations in their graphs and view annotations that other users have made. A key concern in all of these features is to guarantee scalability, as the files containing the graphs are often too large to fit in main memory.

In this report, we present the project we have worked on for the past quarter. First, in section 2, we give an overview of the product that we developed. We then elaborate on the software engineering dimension of this project in section 3. In section 4, we dive into the set of functionalities we have developed specifically for `Hygene`. To see whether our product actually meets customer demands, we conducted a product usability study. The methods and outcomes of this study are discussed in section 5. In section 6, we evaluate the product in its entirety. This also includes a discussion on which parts failed to meet our own and our customer's expectations and what we can learn from those experiences. Finally, we give an outlook on features and improvements we could implement in the future, in section 7.

2 Overview of the Software Product

Hygene was created to reduce the gap between researchers in the life sciences and the enormous masses of genome data they have to deal with. We want to make the exploration of different genomes intuitive and accessible to researchers. We support opening collections of genomes in the form of a GFA file.

On opening such a file, users are presented with a graph visualization interface, allowing them to scroll through and zoom in on different parts of that graph. This interface mimics a map application to enable intuitive and natural access. Various visual encodings represent characteristics of different parts of the graph, giving users as much information as possible at a glance. There also is a pane, situated at the bottom of the screen, that lets them explore an individual node as well as its metadata.

While letting users navigate through the graph is quite powerful on its own, we have also added bookmark creation and annotation visualization functionalities for ease-of-access. Bookmarks allow users to pinpoint locations in the graph and save them for a later time. Users can also open external annotation files, allowing them to see the annotations they encode appear in the graph.

A crucial non-functional property that played an important role in all of our considerations was scalability. Many GFA files are too large to fit in memory without pre processing. As a consequence, we had to come up with strategies to deal with data structures of this size, while preserving high speed of access.

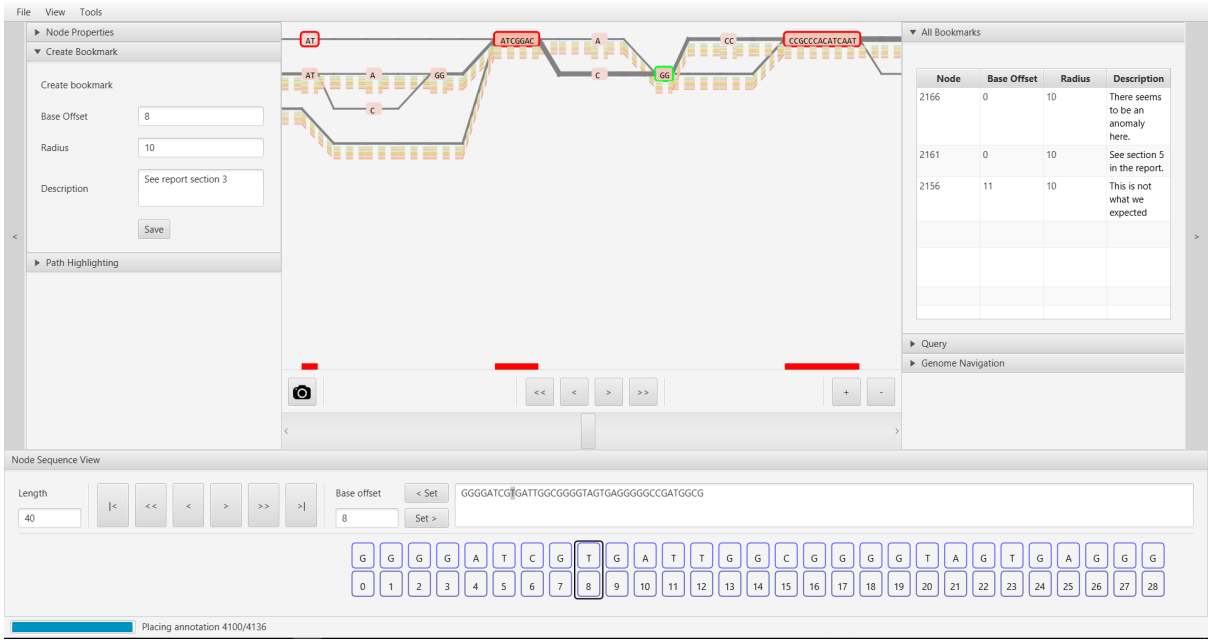

Figure 1: Screenshot of the Hygene application

3 Reflection on the Software Engineering Product and Process

There is a lot more to any software project than its final set of features. While most customers only get to see that final product, an equally important characteristic of that software is its code quality. Among others, this characteristic not only determines how maintainable the product is, but also inherently the ease with which new features can be added. In this section, we reflect on the software engineering side of this project.

We start off by discussing the measures we took to ensure quality of code. Then, we elaborate on our testing approach and the difficulties we faced in the process of testing our application. We conclude this section with a discussion of technical debt and how it impacted our decisions.

3.1 Code Quality

Throughout the project, we paid close attention to the quality of our codebase. This was done through a combination of automated and manual reviews.

3.1.1 Automated Reviews

The automated workflows we had in place consisted of dynamic and static analysis tools. On the dynamic side, we used the Travis continuous integration service to verify that our builds passed in an isolated, clean environment. This caught quite a number of basic errors before they could reach the `master` branch and, although sometimes taking longer than wanted, was a valuable check in our workflow. Next to basic test execution, we relied on Codecov for test coverage annotation. This helped us find a good number of conditional branches that were overlooked during testing at the PR review stage.

We added a number of standard static analysis tools to our development pipeline for static code quality assurance: SonarQube (hosted on a private server for integration with PRs), Checkstyle, FindBugs, and PMD. All of these acted as a second pair of eyes, pointing out mistakes before they could persist. A perhaps less conventional choice was the decision to include the 'Checker Framework'[1] in our build process. This framework formally verified our code to assure that it was not possible to get `NullPointerException`s on running it. We had mixed experiences with this tool: For the majority of the code we found it to provide useful feedback on most of the code, and we indeed noticed a drop in NPEs during runtime. Some parts of the UI code, however, were less compatible with its verification mechanisms, due to the custom object injection strategies they used. We chose to disable the framework on those parts, but this was of course not the ideal solution, because it meant we had definite assurance of soundness for only a part of the project's codebase.

3.1.2 Manual Reviews

No matter how many of these automated tools we set up, however, nothing truly paralleled the added value of manual reviews of proposed changes. We have opened a large number of these 'Pull Requests' during this project[2]. On every PR, we required at least two reviews, and it was not uncommon for substantial PRs to have extensive discussions on them. This practice proved itself very useful in a number of regards: It improved our feeling of shared code ownership; revealed points of improvement in the design of the code that the original author was not aware of; and made us more aware of what the others were working on and how they implemented it in the codebase. On the other side, we might have even gone a bit

[1] https://checkerframework.org/

[2] 420+ at the time of writing this report.

too far with these reviews, since there was a substantial number of comments on Javadoc grammar and style. While it is good to have consistent documentation, we could have spent less time on perfecting it and more time focusing on the improvement of actual code.

3.2 Testing

As our project was written for the JVM, we chose the recently released JUnit 5 as a testing framework and Mockito as mocking extension. During the project, we maintained a reasonable level of coverage, although it inevitably dropped during times where many features had to be implemented in little time. The main difficulty that quickly manifested itself lay in properly testing UI code. Many unexpected obstacles popped up during our attempts at testing code that relied on the JavaFX framework, ranging from uninitialized FXML references, to getting our tests to run on continuous integration, to the inherent difficulty of programmatically simulating user input.

3.3 Balancing Technical Debt

Any substantial project with a limited timeframe carries the dilemma of balancing features against refactors with it. This project was no different, and there were a number of points during the last quarter where we had to ask ourselves that very question: To implement or to refactor? We experienced first-hand that doing the former too often without doing the latter can come back later in the form of an increased difficulty to add new features. While this is a clear conclusion we can draw, we also struggled to see how we could have balanced the two more effectively. The customer had arguably many feature requests, and there was little time to refactor in parallel to those rapid requests.

We also had an encounter with the problem of premature optimization. During the design of an efficient, performant data structure, we neglected to provide the right level of abstraction around it. This was great for performance, but quickly resulted in an increased difficulty to add new features. We had to refactor the data structure mid-quarter, slowing down development on other aspects. In hindsight, it is now even clearer to us that optimizing code before actual barriers are encountered is rarely a good idea.

4 Developed Functionalities

In this section, we discuss the set of features we have developed for `Hygene`.

4.1 Visualizing Large Genome Data

The application starts on an empty screen, with a menu bar at the top allowing the user to open a GFA file. Recently opened files appear in a dedicated sub-menu for quick access. As soon as this file is opened, we perform a series of preprocessing operations to be able to display this file. This preprocessing pipeline consists of parsing the file and converting it to our own data structure, which we cache in a file database next to the original file. Because these operations tend to require a bit more time on larger files, a progress bar is displayed to keep the user updated on the progress.

As soon as the file is fully loaded, we display a central region of the graph in the main graph interface. The graph we are dealing with here consists of nodes (represented by rectangles) and edges between those nodes (represented by lines). This subgraph is lay out in a horizontal fashion, with a combination of the Sugiyama layering algorithm and Barycentric edge crossing reduction. The user can now interact with the graph, in a similar way in which he/she would navigate through a map application. Scrolling lets the user zoom in and out, and dragging moves the graph in the horizontal direction.

There are a couple of visual aids which aim to give the user a better feel for the data. If the user is zoomed in far enough, the first few bases of the base sequences of each node are displayed on top of that node. The colour of the node is also determined dynamically in compliance with a given encoding which can be chosen in the settings. Finally, special types of mutations (SNPs, to be precise) are automatically collapsed and visualized with a special symbol. These mutations are quite common in these kinds of graphs, and indicating them as such improves the user's oversight of the data.

4.2 Working with Genome Graphs

Now that we have a basic toolkit for navigating around such multiple genome graphs, the actual functionality begins. When the user clicks on a node, a side-menu is populated with the metadata of that node. A pane at the bottom allows the user to scroll through the sequence of that node, base-by-base.

To make navigation and search even easier, the user can also navigate through specific genome coordinate systems by selecting a genome and entering a base offset in that genome to which to go to. If the user doesn't know that precise number or wants to perform a more systematic search, he/she can also conduct a RegEx search query on the sequences of all nodes in the graph, which are then highlighted with a special outline. Finally, the user can also choose to highlight one or more certain genome(s), which are represented by paths through the graph. These are then shown in the graph with a colored line across nodes and edges belonging to the selected genome(s).

4.2.1 Creating and Viewing Bookmarks

Just as it can be useful to add post-it notes to different pages of a textbook, `Hygene` allows users to add bookmarks to locations in their graph. These bookmarks consist of a location, being the node to be marked, and a radius, denoting the scale of the region of interest. Optionally, the user can also add a description to each bookmark, to make the tag more informative when visited again later.

All bookmarks can be viewed in a dedicated pane on the right-hand side of the application. To go to a certain bookmark, the user can double-click that item in the list. The node that was bookmarked is then moved into view and highlighted, and the radius that was stored initially is used to automatically set the zooming level to show the entire region of interest.

4.2.2 Viewing Annotations

We also support opening a so-called GFF annotation files in our application. On opening such a file, the user is asked to provide the genome this annotation should be linked to. Once this has been done, the annotations are added to the visualization. All annotations appear on the nodes themselves, in the form of coloured line (different colours represent different annotations). If users want raw search access to the annotation text fields themselves, they can also query these fields with a simple RegEx query entered through a text field.

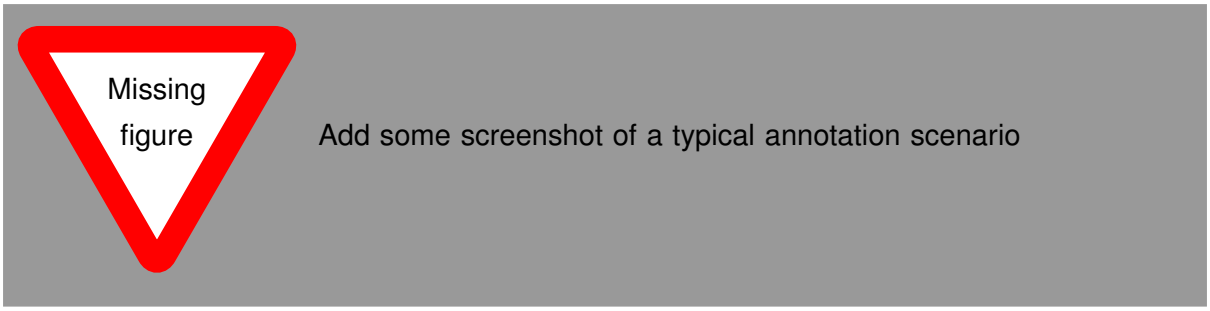

Figure 2: Screenshot of an annotation visualized in the `Hygene` application

5 Interaction Design

Software that interfaces with a human has little use if it is not easy and intuitive to use. Our product `Hygene` is no exception from this: We aim for this product to bridge the gap between researchers in the life sciences and the large amounts of genome data they have to analyse. Carefully evaluating the interface between our users and our software is especially important for this use case. While we as developers have conducted quite a number of manual tests, we are not part of the target audience. To find out if the application actually is usable and intuitive for end-users, we conducted a usability study.

In this section, we first elaborate on how this study was carried out. We then present the results we gathered. Finally, we discuss these results and their impacts, as well as possible solutions to issues that became apparent during the study.

5.1 Method

Our study is carried out with three persons. One is a member of the Broad Institute (US), referred to us by our project supervisor. The other two are individuals familiar with the domain, here in the Netherlands. These two were added because we wanted to increase our test audience. Due to the limited scope of such a project within the study programme, it is very difficult to get feedback from actual researchers that use these kinds of tools daily. Instead, we tried to contact people that are familiar with the domain.

For all three of our test persons, we offered two choices: Either a live interview via Skype (roughly 30 minutes), or a written list of assignments and questions which they can fill in at their own chosen time. The assignments given in written and through Skype are exactly the same, only that we walk the person through the process if they choose the live interview option. In the case of a Skype interview, we ask the person to share his/her screen with us. If they give us their consent, we also record this session to be able to later analyse it again, with the restriction that we permanently remove the footage as soon as possible.

5.2 Results

Unfortunately, the interviews are still to be conducted. We will add the actual results of our usability study to the final version.

5.3 Discussion

Due to the results not being available yet, it is of little use to discuss them, at this point.

6 Evaluation of the Product

Looking back on where we started two months ago, we have come quite far in developing a feature-rich application for multiple genome visualization. As in any software project, there are features that still need improvement to become production-ready. In this section, we take a look at the final product, as a whole, and the challenges that went into creating its features.

6.1 Visualizing Large Genome Data

The main functionality we needed to support for this application is the visualization of large GFA files. We spent a good amount of time on designing the right backing structure to be able to keep such large graphs in memory (when the original file itself simply did not fit in memory). While there were some issues concerning the abstractions we built around it (see section 3), we did manage to support even the largest file that was provided to us and still allow for responsive navigation.

Laying out a graph like this was a challenge on its own. Our initial decision was to develop our own graph layout algorithm, which we tried to do in the first weeks. While we managed to create a working implementation of this, it was nowhere near the quality of layout it needed to have. As a consequences, we decided to start from scratch and use an adaptation of an existing algorithm, which was much better suited to fit this situation. An important lesson that we learned from this episode was that other, existing solutions to a problem should always be regarded with more care during the initial design process, and not be dismissed too easily. As it turned out, that other algorithm was indeed a viable solution to our problem, meaning that we would have saved ourselves time if we had gone with it, from the start. However, this is easier to assess in hindsight than when you are at that spot in time.

Providing navigation means was also not a trivial task: We went from very chunky zooming functionality to a smooth, interpolated version that is now present in the application. A feature that would still need more work on to perfect would be horizontal panning. At the moment, this moves the viewport one entire node further on panning, but ideally, this should also become a smooth navigation control. We have not yet implemented this is yet, however, due to the many other features that awaited implementation.

6.2 Viewing Genome Annotations

A central challenge in implementing the visualization of genome annotations was accessing the genome coordinate system in an efficient, fast way. Annotations are defined to start somewhere in a genome and end at another place. These locations, however, are defined in terms of bases from the start within that genome. This is not data that is easily accessed by a single read, but requires an access to every node in the graph, in a naive implementation. Some sort of index is needed to make accesses faster. Initially, we generated an index stored in the local SQLite file database that resides next to each GFA file. As quickly became apparent, however, both writing and reading from that database was too slow for our use case (loading all annotations), meaning that we had to come up with an alternative solution. We now generate this index dynamically, on-demand, and store it in memory. This has brought a down index and query times by a significant amount, meaning that we now were actually able to load large annotations efficiently.

7 Outlook

Over the past weeks, we have created an application that could be used by researchers in the field. Needless to say, the horizon of what features to add in the feature is very broad. This section will discuss three main themes out of the countless possibilities.

7.1 Creating Custom Annotations

Currently, the user can only view annotations that others have already created. Having the possibility to annotate the graph while in the application opens up a number of possibilities for users and provides a more complete user experience. These annotations could be added manually through text input fields, or visually by clicking on certain parts of the graph and having the program identify which parts to annotate.

7.2 Online Integration

There is a growing trend nowadays towards cloud-powered applications. We envision `Hygene` in a similar ecosystem, providing sharing, uploading, and downloading functionality for graph and annotation files. We could maintain an online library of GFA files, allowing users to easily download and view files that others have uploaded. Users could also share bookmarks and annotations with the click of a button, or even collaboratively annotate the same graph, in real-time.

7.3 Tutorial Material

Any application which satisfies a niche demand, and is thus to be used by experts, needs some form of education material to get users up to speed with how things work. `Hygene` is no different: We cannot reasonably assume that every functionality is instantly clear to new users. A simple way to address this is to show a tutorial at the first start of the application. This tutorial could explain common functionalities, thereby gently introducing the user to our application. It could even be run at the same time as the loading takes place of larger files, saving the user time that would be spent waiting otherwise.

Glossary

genome the haploid set of chromosomes in a gamete or microorganism, or in each cell of a multicellular organism. 2, 3, 6

GFA a tab-delimited text format for describing a set of sequences and their overlap. 3, 6, 9, 10

GFF a file format used for describing genes and other features of DNA, RNA and protein sequences. 7

mutation the permanent alteration of the nucleotide sequence of the genome of an organism, virus, or extrachromosomal DNA or other genetic elements. 2

References

NHGRI. (2003). Human genome project. National Human Genome Research Institute. Retrieved from https://www.genome.gov/11006943/human-genome-project-completion-frequently-asked-questions/