[DRAFT] Emergent Architecture Design

Group PL4

10 June 2016

Arthur Breurkes	4398033
Ricardo Jongerius	4379500
Niek van der Laan	4296915
Daphne van Tetering	4375165
Niels Warnars	4372069
Ties Westerborg	4386817

Context coordinator:	Dr. T. Abeel
Teaching assistants:	Mr S. Huisman MSc
	Mr J. Linthorst MSc
	Mr J. Hommes

Table of Contents

1. Introduction

This document serves to describe the architectural design of the genome visualizer that will be built for the Programming Life context project.

1.1. Design goals

During the development of the DART-N visualizer, a number of design goals have been specified. These design goals have been dafted to retain a level of quality.

1.1.1. Usability

The DART-N visualizer aims at users that are considered experts in the field of genome research. For this target audience only a limited set of tooling is available to visualize and compare mutations in data sets consisting of hundreds of genome samples. Most tooling is only able to visualize single genomes or does not have the ability to show mutations in a graph like setting. It is therefore important that the visualizer provides the correct functionalities that satisfy the needs of the customers. A consequence of this specific target audience entails that using the visualizer may not be as straight-forward for a layman as it is for an expert.

1.1.2. Scalability

The visualizer is initially going to be designed for a small data of ten genome samples that will be used to finetune all fuctionalities and optimize the graph processing. The final goal is to achieve adequate scalability so that datasets of multiple hundreds of genome samples can be processed and analyzed. More specifically, we look to visualize the data in an efficient way, by only showing relevant information with regards to the zoom level.

1.1.3. Maintainability

While implementing the customer wishes, a high level of maintainability is tried to be maintained. To do so, the aim is to keep code well documented and structured in a clear way while being covered by as many test cases as possible. On the short-term this has as advantage that whenever a bug or defect does occur, it will be easier to locate and fix it. In the long-term high maintainability aids future development of the product, even for programmers who were not involved in the project.

2. Software Architecture Views

In this chapter, we look to discuss a sketch of the architecture of our system. To do so the first the decomposition is covered followed by the core functionalities and an overview of things that can improve code quality.

2.1. Package lay-out

Classes are stored in packages to keep everything ordered. The following packaging structure has been specified:

- Application - Contains the GUI functionality
    - Controllers - Contains the controllers and factories
    - FXObjects - Contans the individual GUI elements
        - Graph - Contains the graph node cells
        - Lay-out - Contains classes responsible for the tree / graph placement
        - Tree - Contains the tree node cells
- Core - Contains core classes like the Node, Graph and Tree classes

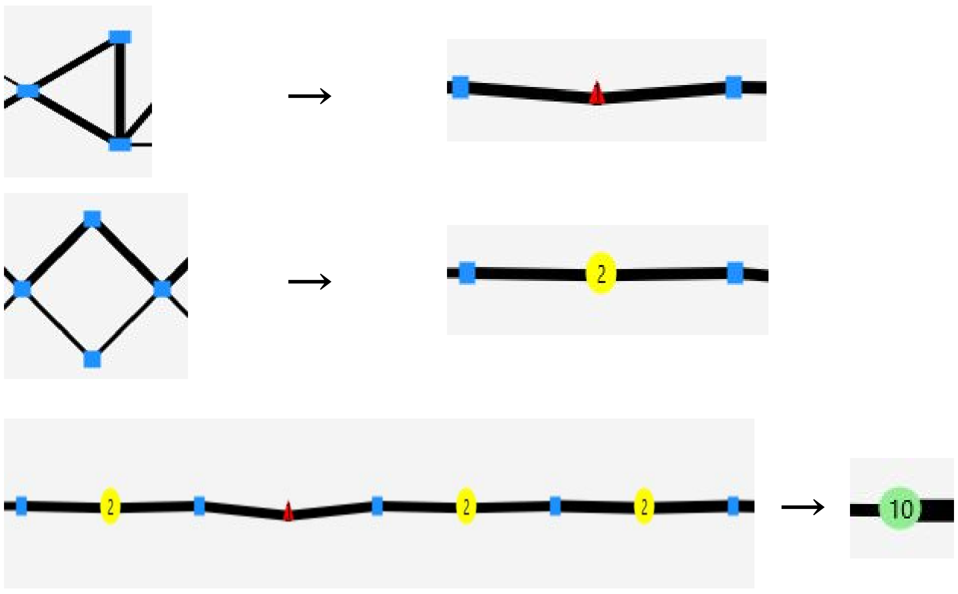
2.2. Core functionalities

A number of functionalities is most imporant for a proper usage of the visualizer, these features include:

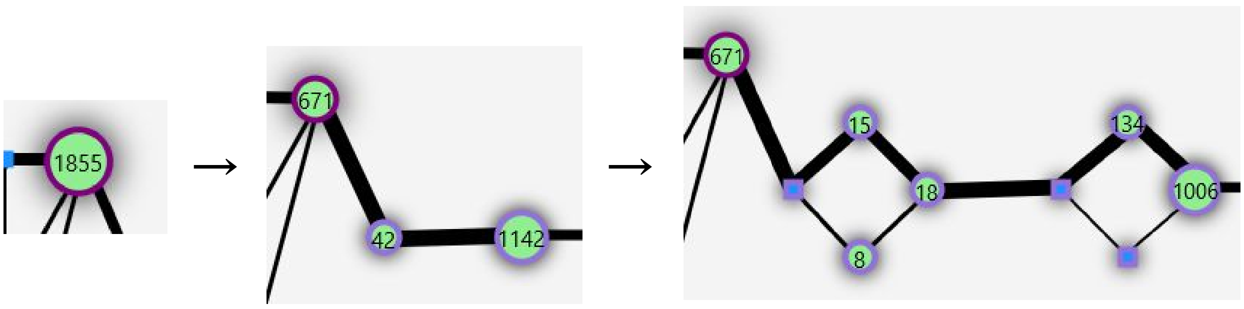
- **Parsers**
  A user has the ability to load the following data from file:
    - GFA file containing genome data of multiple samples.
    - NWK file containing the phylogeny of the data set.
    - GFF file containing annotation data.
    - XLSX file containing metadata of the data set.

  On selection a file’s data is read from disk and converted into a useful data structure that can be easily processed and queried.

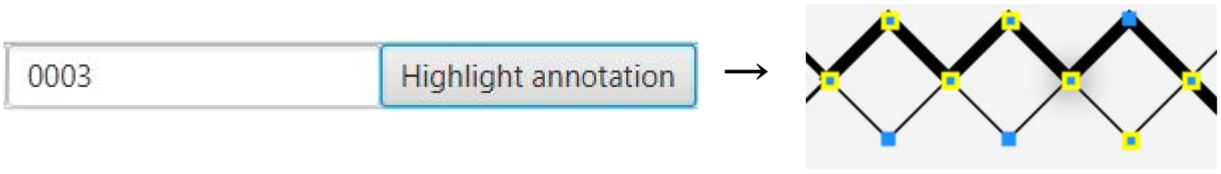
- **Graph reducer + visualization**
  Visualizing the entire graph - which can contain up to several hundred-thousands of nodes - is not feasible. Therefore bubble and indel mutations are collapsed into special bubble or indel nodes. If multiple nodes are spanned horizontally they can be horizontally collapsed into a collection node. This way an entire graph can be shown collapsed without having to scroll horizontally or vertically. Each collapsed node can again contain multiple other collapsed nodes. This continues collapsing can be applied till no more collapsing can be applied effectively creating multiple collapsing level maps through which a user can scroll.



- **Practical semantic zooming**
  The collapsed level maps genereted by the graph reducer can be scrolled through as shown in the figure below. Whenever a user selects a node and zooms, in the left most underlying bubble will stay in screen while the other nodes forming the original upper bubble keep highlighted.

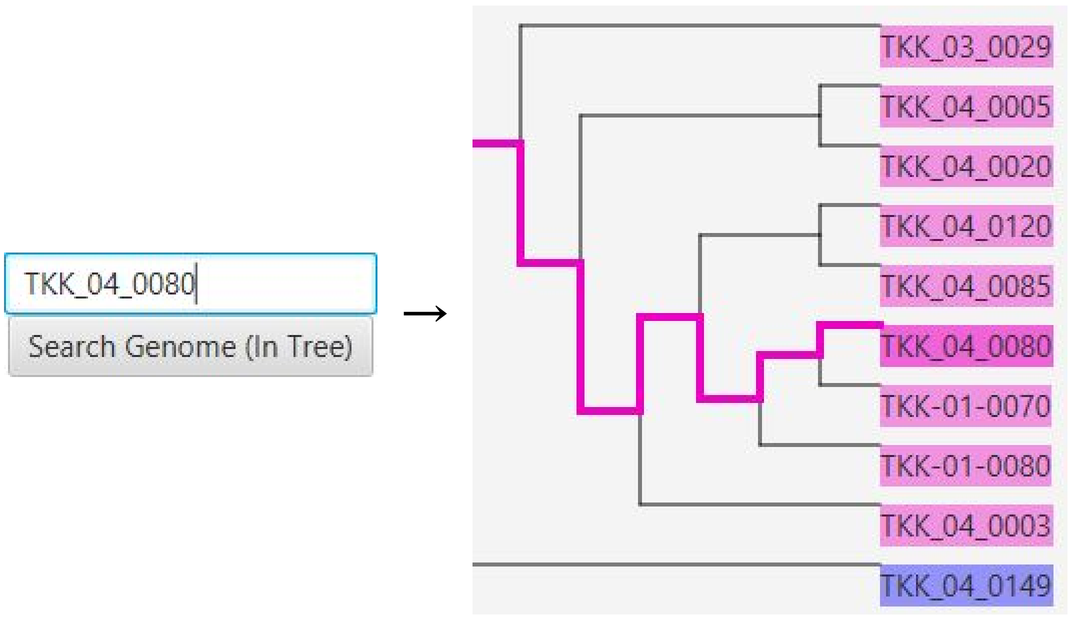


- **Annotation highlighting**
  If an annotation (GFF) file is loaded from disk the user has the ability to select / highlight an annotation specified for a reference string. An annotation can span multiple nodes and a node can be part of multiple annotations.



- **Phylogenetic tree with search / filter options**

The phylogenetic tree shows the evolutionairy relationship between genome samples based on single nucleotide mutations. For easy navigation the user can select one or multiple genomes in the tree which can also be filtered in the graph.



2.3. Persistent Data Management

The program will deal with different data sets, such as the phylogenetic tree, the genome sequence alignment graph, the metadata and the annotation data. These are used to generate node and edge data that is stored in memory. This data should be available to the application at all times during runtime, so no overhead is introduced in on the fly loading of additional data from disk.
Data that will be written to disk only involves the names of recently loaded data files.

2.4. Code Quality

To maintain a high quality of code, we look to do the following things in particular:

- **Pull-Based Development**
  During all times, new features are developed in separate branches. When someone considers a feature as done, he or she can make a pull request. When the request gets approved by at another team member, the branch can be merged into the master. This way a reliable master branch is ensured at any given time.

- **Testing**
  To ensure the correctness of the code, a test coverage of at least 80 percent is demanded. JUnit has been chosen as the main testing framework. Other frameworks, such as Mockito and Maven, are used aswell.

- **Analysis**
  Last but not least, we look to code according to common conventions. To support this effort, well known tools as CheckStyle, FindBugs and PMD will be used.