[DRAFT] Emergent Architecture Design

Group PL4

29 April 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

Before implementing the DART-N visualizer, a number of design goals have been specified.

1.1.1. Component independence

One of the main aspects of the application is that the Graphical User Interface should be independant from the model. As a result of this separation, the GUI and the back-end should be able to collaborate together, but must be able to function separately as well.

1.1.2. Usability

The DART-N visualizer aims at users that are considered experts in the field of genome research which only have a limited set of tooling to visualize and compare mutations in data sets consisting of hundreds of genome samples. It is therefore important that the visualizer provides useful functionalities that satisfies 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.

1.1.3. Scalability

The visualizer is initially going to be designed for a small data of ten genome samples, but 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.

1.1.4. Maintainability

While implementing the customer wishes, a high level of maintainability is tried to be maintained. To do so, well documented code and test cases are necessary at all times. This way, whenever a bug or defect does occur, it will be easier to locate and fix it. Another advantage of high maintainability is that any possible future development of the product, even for programmers who were not involved in the project, should be relatively straightforward.

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 hardware/software mapping and an overview of things that can improve code quality.

2.1. Subsystem decomposition

To correctly manage the visualizer, it is decomposed into smaller subsystems in which each subsystem had its own functions and responsonsibilities. The DART-N visualizer can roughly be decomposed in the following components:

- **Parsers**
  The parsers read the genome data from the provided phylogenetic tree and genome graph, and convert it into a useful data structure that can be easily processed and queried.

- **Database**
  To store all our parsed data, a database will be used. This way a more than linear increase in the number of genome samples can be managed without running out of CPU and memory resources.

- **Graphical User Interface**
  The Graphical User Interface should provide the user with an intuitive and detailed overview of the dataset. At all times, the user should be able to tell what part of the dataset is being looked at, and can be influenced in a multitude of ways in what way to research the dataset.

2.2. Hardware/Software Mapping

As discussed before, the visualizer looks to make use of a database to ensure scalability. However, the back-end database should also be easily swappable.

2.3. Persistent Data Management

The program will deal with different data sets, such as a phylogenetic tree, the genome sequence alignment graph and the metadata. The database in which this information is stored, should be available to the application at all times during runtime, so no overhead is introduced in on the fly loading of data from disk.

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 least two other team members, 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.