PROGRAMMING LIFE CONTEXT
DELFT UNIVERSITY OF TECHNOLOGY

DNA is Not an Acronym

Architecture Design

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

May 2017

Contents

1 Introduction

In this document we give an overview of the structure of the final product. The product itself is a genome visualisation tool. Data is formatted and stored as a GFA file. The aim of the application is to allow researchers to quickly analyse large amounts of data in a visual manner, draw conclusions and share these findings with others.

A GFA file consists of Segments, Links, Containments and Paths. For our purposes, we only deal with Segments and Links. Segments represent sequences of base pairs, and links are used to connect different segments. These GFA files can be used to encode genomes, and is why this application must deal with these files.

1.1 Goals

The overarching goal of producing an application can be split up into multiple goals. Firstly we must consider the performance of the application, and second the reliability.

1.1.1 Performance

An incredibly large amount of information is encoded in genomes. That in turn means that the application must deal with this data.

1.1.2 Reliability

Our application will be used by researchers. An unreliable application can lead to unreliable conclusions, leading researchers to discard our application. As such, the application should accurately display information.

2 Software architecture views

2.1 Subsystem Decomposition

`Hygene` is a native application, meaning it runs directly on the users machine. Several subsystems are present within the application:

2.1.1 Graphical User Interface (GUI)

The GUI provides the user with the capability of interacting with the application. The GUI will take care of presenting the user with the appropriate views and delegating events to the appropriate subsystem. As an example, opening a file should start the `Parser` and consecutively send the parsed data to the graph visualizer.

2.1.2 Graph Visualizer

The graph visualizer is a view embedded into the GUI. However, because of its importance, it is considered a subsystem on its own. The graph visualizer's primary purpose is to draw the graph on the screen. This graph is completely built up from primitive shapes to minimize the overhead that would be introduced by using a third-party graph library. Finally, the graph visualizer implements the algorithm needed for determining the layout of the graph. This is a challenging topic, since each node has to be aligned such that the graph conforms to a layout which is intuitive and aesthetically pleasing.

2.1.3 Parser and Data Structure

The Graphical Fragment Assembly (GFA) format is a well-known specification for describing sequence graphs. The parser is responsible for parsing GFA files and transforming them into `Hygene`'s internal data structure.

2.1.4 Data Storage

While using `Hygene`, the user might want store meta data such as settings or bookmarks. Section 2.3 describes this module in detail.

2.2 Hardware-Software Mapping

`Hygene` is tailored towards desktop devices. As such, the hardware it will run on is quite restricted. We do not plan on dividing the execution into multiple processes, since we're relying on threads for concurrency. The DNA visualizer will run locally, requiring no communication with other computers or processes.

2.3 Persistent Data Management

The application requires very little data to be stored persistently. Since the amount of data is very small, it is not wise to use a (for instance SQL) database. This is simply not needed and gives too much overhead. So, we have chosen to store data persistently using JSON-based text files.

The following items will be stored persistently:

- History of up to 10 recently loaded GFA files.
- Custom bookmarks the user placed in a graph.
- Last position within a graph before exiting the application in order to restore the state after a restart.

However, the input of the application, GFA files, are loaded from the local file system by the user of the application. Since these files can be quite large, we have to read them from disk in an efficient way.

2.4 Concurrency

`Hygene` is not a heavily concurrent program by design. Concurrency will only take place in two parts of the program: in the UI and in the data structure.

The UI must remain responsive even when the program is parsing or processing a very large file, or when parts of the graph are being processed. The UI must show an animated loading icon or bar to indicate that it has not crashed, and the user should be able to cancel the current operation.

This type of concurrency requires communication between the processes: The UI must be aware of the progress of the loading action, and the UI must be able to send messages to the other thread to cancel it. This communication can be implemented using methods, as calling a method on another thread is a way of sending a message.

The data structure should be able to handle very large amounts of data, and this handling can be sped up using concurrency. Multiple threads could parse different parts of the file or analyse it for certain properties. This type of concurrency sees the individual threads perform similar operations on different pieces of the same data. As long as the different threads access different parts of the structure, concurrency should not be a problem.

Neither type of concurrency uses shared resources, which means that deadlocks are impossible.

3 Code Quality and Testing

In this section we will discuss the code quality and testing practices we have implemented within our team. First we will describe the value of code quality and which methods, such as static analysis, Continous Integration and pull-based development, we use to ensure a high level of code quality. Then, in section 3.2 we will discuss our testing work flow and which tools we use to write intuitive tests.

3.1 Code Quality

Within our development team we consider ensuring high code quality very important. A high code quality makes applications easier to maintain, less costly to write extensions, easier to understand for newcomers and better testable. So, there is value in it for both developers and stakeholders.

3.1.1 Static Analysis

A lot of possible areas within our application with lower quality we can spot early during development, because of the five static analysis tools that we use: CheckStyle, Checker Framework, FindBugs, PMD and SonarQube.

CheckStyle guarantees a consistent code style throughout the applications. This means the code formatting will always look familiar to other programmers of the development team, making it easier to understand the code, spot potential bugs, and to extend the code. We have created a rule set based on the SUN style rules, with a few changes to better reflect the code style opinions of the team. Besides using the Integrated Development Environment (IDE) plugin, we have also included CheckStyle in our Gradle build flow.

We are using the Checker Framework to guarantee (by formal verification) that our application cannot produce NullPointerExceptions. This helps spotting a lot of unforeseen edge cases that are often relatively easy to solve. By providing simple annotations, we can omit a lot of manual null-checks, making our code much more readable.

FindBugs does what the name suggests: it is a static analysis tool that will search for potential bugs within your code. For instance impossible casting or testing for floating point equality. We are using the default FindBugs rule set. Besides using the IDE plugin, we have also included FindBugs in our Gradle build flow. For FindBugs our builds will even fail in case it detected an error.

PMD, the Programming Mistake Detector, will catch erroneous parts of our applications such as unused variables or empty catch blocks. We are using the default PMD rule set.Next to the IDE plugin, we have also included FindBugs in our Gradle build flow. For FindBugs our builds will even fail in case it detected an error.

SonarQube does more or less all of the above: it finds potential bugs and vulnerabilities, it looks for code smells, it shows code coverage progression and will report on duplications. We are using the SonarLint IDE plugin to detect this,

but have also set up a Continous Integration server for it. We will discuss this further in the next section.

3.1.2 Continuous Integration tools

We have used a few Continous Integration tools to ensure our application is in a good state before adding a new feature or bug fix. For building the application, executing our tests and running all static analysis tools we are using Travis CI. After the Travis CI builds have executed a coverage report will be uploaded to Codecov, which gives us feedback on the progression of code coverage. It does this by providing reports, but also by giving comments on Pull Requests. Finally, we have also set up our own SonarQube server to provide feedback within Pull Requests and to give us easy accessible reports about the current state of our application.

3.1.3 Pull-based Development and Code Reviews

For this project we have adopted the pull-based development model. Concretely, this meant that code was only added or removed through Pull Requests, and therefore was developed on its own branch. Opening a Pull Request will automatically trigger our Continous Integration tools (see previous section) and provide us with their feedback.

More importantly, Pull Requests allow us to easily review each others code. As an internal rule, we have made it mandatory that every Pull Request must have been reviewed by at least 2 other team members. Code reviews will help us finding mistakes in implementation, identifying potential bugs and catching inconsistent styling (not caught by checkstyle). It also helps to improve everyone's understanding of the code base.

3.2 Testing

Automated testing is considered vital to writing a successful application by our team. We are using modern test tools such as JUnit 5, AssertJ, and Mockito to write intuitive tests. Whenever possible, we try to write tests at multiple levels (unit, integration, end-to-end, acceptance). To this end we are using TestFX to test our UI on a mix between end-to-end and acceptance level.

We are using Jacoco in our Gradle builds to generate code coverage reports. These reports include line coverage, which will be used for grading purposes. However, internally we like to use Codecov's metric, which takes both line and branch coverage into account.

Glossary

Continous Integration frequently integrating code in shared code location. 5, 6

GFA The GFA format is a tab-delimited text format for describing a set of sequences and their overlap. 2

Gradle a build system for Java application, similar to the Apache Maven system. 5, 6

IDE Integrated Development Environment. 5

Pull Request a request to integrate code in a shared code location. 6