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

June 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. The application should allow the user to be able to navigate quickly through the large amount of data without hassle and delay.

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 and reliably allow navigation of very large multiple-genome graphs.

2 Software architecture views

2.1 Subsystem Decomposition

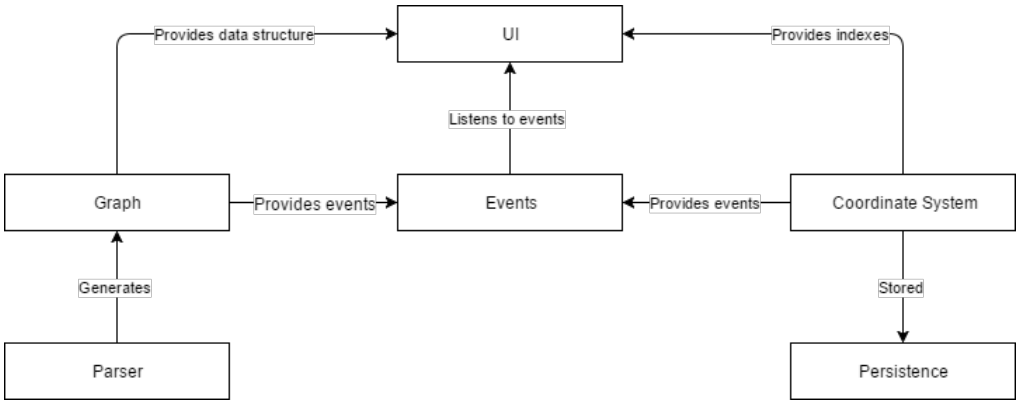

Figure 1: A global overview of the current application structure.

`Hygene` is a native application, meaning that it runs directly on the user's machine. Several subsystems are present within the application.

Figure 1 gives a global overview of the interaction between the application's subsystems.

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. To achieve this, the UI consists of several modules.

The settings menu allows the user to customize the application to their needs. It allows the user to adjust properties of the graph such as its colour scheme, to change the logger level, and to jump to specified points in the graph.

The node properties view shows the properties of the node currently selected by the user. It gives the number of neighbours, its sequence, the annotations it is part of, and a preview of the node in isolation.

The node sequence view displays the selected node's sequence on-screen, making it easier for the user to scroll through a node and select base offsets for bookmarks.

The user can also create and view bookmarks, which are stored in the application's database. The user can create bookmarks by clicking on a node and entering the offset within that node and a description. When the user loads a

bookmark, the application jumps to that node and highlights it to make it recognisable.

2.1.2 Graph Visualisation

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 dimensions calculator converts coordinates from the data structure into pixel coordinates. It updates the cached nodes each time a new centre node or radius is set based on the nodes given by the `GraphQuery`.

The graph visualizer takes nodes, calculates their coordinates using the graph dimensions calculator, and then asks drawing toolkits to place them on a canvas using JavaFX primitives.

The graph movement calculator waits for the user to scroll or pan, changes the size of the viewport, and indirectly triggers the graph dimensions calculator to adjust the centre point query's radius and centre node.

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.

The main challenge here is to avoid the use of objects, since that involves significant overhead. Loading nodes dynamically into memory also induces overhead, so we aimed for a storage format that facilitated storage of all nodes.

To address both points, we created a data structure that consists of only primitives as elements of a 2-dimensional array. Each level-1 array corresponds to a node, having integers for different components:

- node id
- line of the file where this node resides
- colour indication
- positioning algorithm indication

2.1.4 Data Storage

While using `Hygene`, the user might want store metadata 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 relatively small, we decided to use databases and the AppData folder for file specific and general persistent data.

The history of ten most recently loaded GFA files is something that can be stored irrespectively of the loaded GFA file. Therefore this data is stored in the AppData folder.

Bookmarks are associated with a specific GFA file. These are therefore stored in a database which is created alongside the GFA file. We also store a hash of the file in the database to verify that the file has not been tampered with. The database version is also stored in the database itself. This is used to verify that it corresponds with the database version the application expects. If a discrepancy is detected, it means that the user is most likely running a newer version of the application, and the database is rebuilt to prevent and possible undefined behaviour.

However, the input of the application, GFA files, can be quite large. To make this data manageable, we first convert it to an internal data structure. This is then written in its entirety to the database, allowing quick loading of the genome the next time the user chooses to load this genome.

2.4 Concurrency

`Hygene` is a program that utilizes concurrency in several different components of the program which include but is not limited to: the graphical user interface, the data structure and computations upon it, and retrieval of the metadata. The graphical user interface (GUI) should always be responsive even when the program is parsing or processing a very large file, performing computations upon the graph, or metadata is being retrieved through dynamic loading. For the latter reason, our GUI thread will delegate all operations that are not possible to computed within millisecond to different thread.

The data structure is the core of our program and is the subject of computation done by subsystems. An example would be (re)computing the layout whenever the user zooms. Making the latter work efficiently was a bit more tricky than expected. When the user scrolls we want the layout to update as smoothly as possible; therefore we start the computation as soon as possible. However, this computation is not done asynchronously since it is not instant and cannot be done on the GUI thread. However, if we were to (re)compute the layout for each scroll event a large amount of threads would be scheduled, of which only the last one would be relevant.

To address this issue, we created a special class that will throttle the number of times an action can be executed. When the throttler is called, it checks if the specified action is already running, and if it isn't, executes it. If an action is

already running, however, it adds it to a queue. This queue has a size of exactly two actions, and the oldest entry in the queue is removed if a new action is added while the queue is full. This way, it is guaranteed that the code execution is throttled while a call will always guarantee that the action is executed *eventually*.

This type of concurrency doesn't use 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. 7, 8

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. 7, 8

IDE Integrated Development Environment. 7

Pull Request a request to integrate code in a shared code location. 8