Architecture Design

Programming Life
Team Dynamites & Butterflies

Jasper van Tilburg - 4394554
Marc Visser - 4318234
Lex Boleij - 4153197
Eric Dammeijer - 4412265
Jip Rietveld - 4486528

23-6-2017
TU Delft

Table of Contents

Introduction

This document describes the architectural design of the multi-genome interactive visualizer. The product was created for the Programming Life context project and serves as an extension of current genome viewing tools.

Design goals

In the design goals section we explain our design goals. These are goals that stay in the back of our head during development of the entire product. These design goals are there to ensure good coding and good teamwork.

Usability

The goal of our product is to be useful. For the customers we want to make sure the application feels intuitive and modern. Our goal to ensure that happens is by the constant feedback loop we receive by the use of the SCRUM principle. With constant usability feedback and the must-have of a working application at the end of each sprint we can ensure this happens.

Maintainability

Our goal is to ensure that the product is maintainable. Due to the open-source nature of our project we encourage others to aid in updating features. To make sure they can quickly aid us and don’t waste too much time on understanding our code we want to make sure it is maintainable. A lot of comments, clear variable names are all ways of making sure this happen. We try to ensure this by reviewing each other's code after a feature is made.

Scalability

Genomes come in all shapes and sizes. The initial runs and products will be built for genomes with ‘only’ 1000 base pairs. The idea is that at the end of the product cycle our application will successfully parse and load genomes up to 10+ million nodes. With that in the back of our mind we want to ensure our application scales well.

Software architecture views

Subsystem decomposition

In our application we have 5 different packages splitting the code into different parts. In the following part we will break down the different uses of these packages and what functionalities are handled there.

Exception

In this package we have a lonely class called NotInRangeException. We had to build this exception ourselves to handle getting the centrenode. We figured this exception was the best description of our error.

Graph

The graph package contains multiple classes with functionality regarding the graph and the different parts of it. The biggest classes are the SequenceGraph and SequenceNode classes. These classes are what the user sees and interacts with the most.

Besides these 2 we have the class Boundary. These boundaries are used to determine the subgraphs we create for the user to see.

Parser

The parser package contains two parsers. A GFA parser that can parse .gfa files into different structures and data files we use to visualise the graph. And besides that there is a GFF parser class that handles parsing the annotation files into a data structure need to visualise them.

GUI

Last but definitely not least we arrive at the GUI package. In the main part of this package we have different important classes. Besides these important classes it contains the sub_Controllers package in which The App class contains all logic and code to get the application up and running and loaded onto your screen. Besides that we have all the means necessary to visualize the graph parsed in Parser and made in Graph. We do that with multiple different classes. GraphDrawer is the main class that eventually draws the graph.

Finally we have the class MenuController. MenuController is the controller of the fxml file that represents the stage the user sees the most when using the application. It handles all button presses on the field as well as typing and different menus. To split the code from logic to front-end MenuController makes use of the sub_controllers package.

In this package a lot of logic is contained surrounding the panning, zooming, recent files, bookmarks etc. These contain the values and algorithms needed to view the graph. MenuController tries and only handle gui elements.

Furthermore we implemented a model view controller pattern, the model consists of the parser and the sequence graph, the controllers handle how the UI works based on the model, and the view updates the model (figure 1). They communicate with each other using a observer pattern. Furthermore the Drawable canvas and graph drawer are singletons, as we only want a single instance of those objects. We haven't made any interfaces, this is due to the nature of our program. It is basically a graph with nodes, which means not many different kinds of objects are used. This makes programming to an interface less useful.

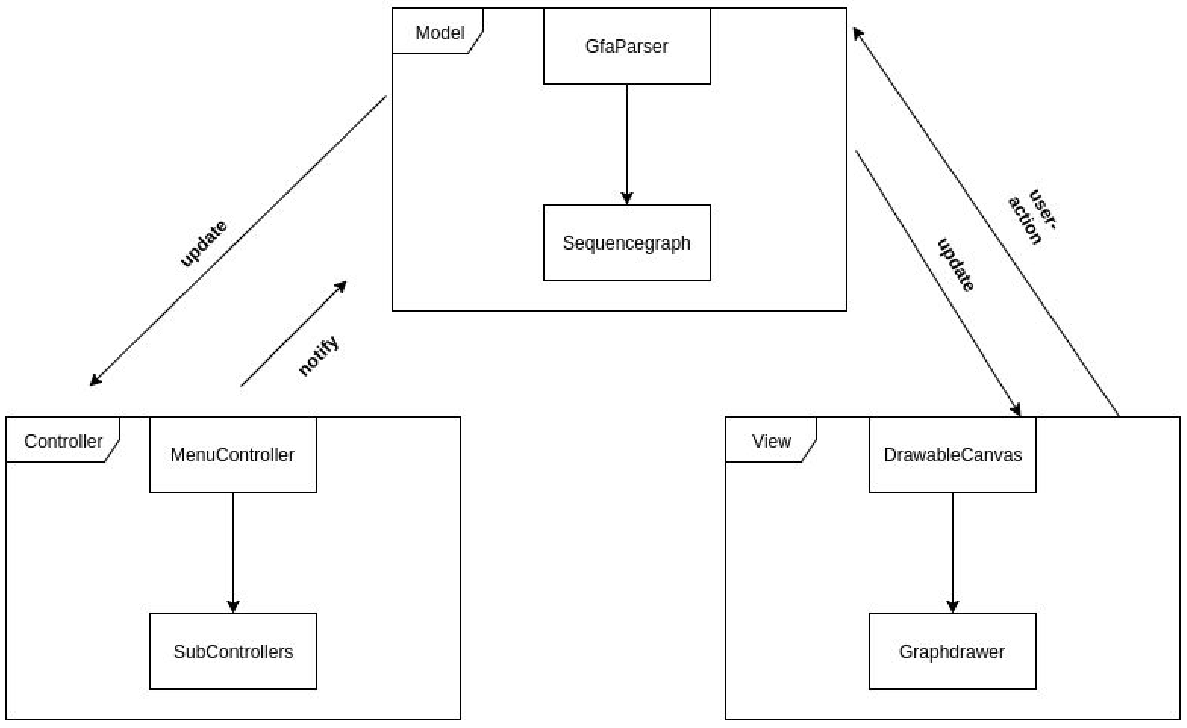

Figure 1. MVC, Observers

Persistent data management (file/ database, database design)

In order to save the sequence data, we make use of a tool called MapDB. MapDB saves the sequence of each node in a HashMap structure. This gives us fast look-ups, negative side is that for large files, the database becomes quite large because of the HashMap (not because of the Strings)..

Managing data was one of the hardest parts of our application. With graphs that can grow to 10 million nodes it's a must to be efficient about everything we were implementing. All these large datasets also meant we could not always store our data in the allocated java memory. This meant we had to make smart use of database files and the differences between hot and cold loads.

At the end of the project we finally create 4 different files per gfa file a user wants to load. A MapDB file, and 3 txt files. called sequence.db, childArray.txt, parentArray.txt and genomes.txt respectively. In the following part I describe why we create these files.

The sequence.db file is made to store the base pair sequences of the nodes. Because these are all strings and can all become very very large sets (up to 58 million characters in total * the amount of genomes) we decided for a map.db file. We only need to query this file once a sequence is asked and a single query is fast enough. Please note that the length of these sequences are essential to the program but because those are merely integers we can safely store those in memory of the subgraph.

The childArray.txt and parentArray.txt files are the edges. Every pair of integers in these files represent the edge from certain node to a different node. These files can grow very large with big datasets which is why we save them in these files.

The genomes.txt file holds a lot of integers in itself. Every new line represents a different node. And every line is then split on ";" and around that on ",". The first part represents the ids of the genomes that go through this node. The second part the relative coordinates. These are only queried when building the subgraph. But too large a dataset to hold in constant memory. Dynamically loading them was an option but we preferred this solution.

Concurrency

Our programme has very little concurrency in it. The only moment we have actual concurrency is when we parse a .gfa file. That is because when we parse the file in one thread we show a progress bar on another thread. Using threads here makes sure you can still use and close the programme even when it is busy doing the calculations for parsing the .gfa file.

Beside the above we have some concurrency for MapDB. We make sure the DataBase is closed every time another file is loaded or the program is closed. This way we prevent overlapping calls to the database.

Glossary

Cold load - the first time loading a gfa file
Hot load - loading the program after the gfa file is already loaded once