Architecture design

Desoxyribonucleïnezuur, context project 2017

Toine Hartman - tjbhartman - 4305655
Martijn van Meerten - mvanmeerten - 4387902
Iwan Hoogenboom - ihoogenboom - 4396634
Yannick Haveman - yhaveman - 4299078
Ivo Wilms - iwilms - 4488466

June 23, 2017

Introduction

This document describes the architecture for the genome viewing program for the programming life context of the context project 2017 (“the program” or “the product”). The way we work, Scrum, is by definition adaptable and favors change, which means that this design is not set in stone. We may change the architecture later if that appears to be a better choice. In that case we will also update this document to reflect those changes.

The program is for viewing multiple genomes, particularly for viewing local differences (“mutations” or “bubbles”) between these genomes. These genomes and bubbles are abstracted as nodes of DNA sequences. Each node links to some other nodes to define the sequence. The purpose of the program is to visualize these nodes in a graph. As these genomes can consist of several millions of nodes, it is infeasible to show them all on screen in a meaningful way. Instead, the user should be able to zoom in on a particular place in the graph to inspect it, move the view (“panning”) to check out the neighbouring nodes of these places, and be able to select nodes to see detailed info.

For the remainder of this introduction we will describe the design goals for our product. In the second part, we will describe the program architecture from four different perspectives. The first one is subsystem decomposition, which elaborates upon the smaller, more or less independent parts of the program. After that, we have hardware/software mapping, which is how our program will map onto hardware. Next, we have persistent data management. This describes the way we store data between runs of the program. Last, we have concurrency, which describes the way we will use concurrency to keep our program as responsive and fast as possible.

Design goals

These are the primary design goals for our product. We will need to find the right balance between all of these goals, so that our product is the best it can be within the given constraints.

Responsiveness

The program should be as fast as possible, as people will be interacting with it in real time. The UI should definitely not “hang”, as this will severely decrease the usability of our program. Loading a file may take some time (a few minutes if necessary, but preferably only a few seconds), but panning and zooming should be “instant”, which for us means it should not take more than 0.1 second. Other actions should also be instant if this is reasonable. Currently, loading or saving a file is the only action that may not be instant.

Code quality

Good code quality is desirable for any software project. We will keep the code quality high by using tools like maven, checkstyle, sonarqube, and Travis CI (continuous integration), and

we will write tests to test the code. We will also try to follow the architecture design outlined in this document as long as that is deemed practical, so that our project consists of few smaller submodules instead of being one giant monolithic piece of code. We will also attempt to fix technical debt as soon as possible to prevent it from accumulating and slowing us down.

Coding process

To achieve the best possible product, we should use the best possible process for making it. As most prominent part of our process, we will be using SCRUM. Each week, we will choose tasks to be done by the end of that week. These tasks will largely consist of features that need to be implemented each week, but may also pertain to documentation (writing this document was a task), organization, or be a miscellaneous task. We will be using git as our version management system, in order to facilitate coding together effectively and to keep track of our history. We will also be using github, which has pull requests as the most important feature. Pull requests facilitate pull based development, which we will use to further keep the code quality high by requiring that all tests pass (via Travis) and a code review (which should catch any bugs or other technical debt) before a merge.

As part of the process, we will be using test driven development, which dictates that we first write tests describing the behaviour of a piece of code before actually implementing the code. This ensures that we have thought about what the code should do, instead of implementing the code and then writing tests based on what the code *does* instead of what it *should* do.

Efficiency

The program should be able to operate on enormous datasets, but still be responsive. This means that it needs to be efficient in its use of resources. There are a few bottlenecks: memory (RAM), cpu time and IO (reading from disk). When opening a file, the bottleneck should be IO. There really is nothing we can do to speed that up, but we can use concurrency to read the file in the background while the user can interact with the part that has been loaded already.
We can manage memory by writing everything to disk in an easily readable format, and only loading those parts that we need right at that moment. We might also load things ahead of time if we have time to implement such predictive loading.
The last bottleneck is cpu time. This should only be a problem when the entire data structure is used at once, like when we would try to filter on nodes matching specific criteria. Even with multiple CPU’s at multiple Ghz, that will still take time when doing some cpu intensive task on millions of nodes. The only way to increase performance is by precomputing more efficient data structures for those searches (for example, have a list sorted on node sequence, a list sorted on sequence lengths, sorted on id, etc). This is a classic example of space/time tradeoff, and we will have to see whether we will implement it.

Performance

The performance of the program is important, because it is the most straightforward way to increase responsiveness (how do you get a faster response? By actually having a faster response). The performance of our program will mostly be measured by how quickly it can progress actions / requests from the user, but it should also be efficient with resources. Efficiency is less important than responsiveness, but that does not mean that we will sacrifice every bit of efficiency for better responsiveness. We will try to find a reasonable balance between the two.

Perceived performance

While we obviously value actual performance, there is no reason not to give the user the idea our program is quicker than it is. If users get the feeling they are waiting less, even if they actually are not waiting less, we decrease frustration, which increases the (apparent) usability of our program.

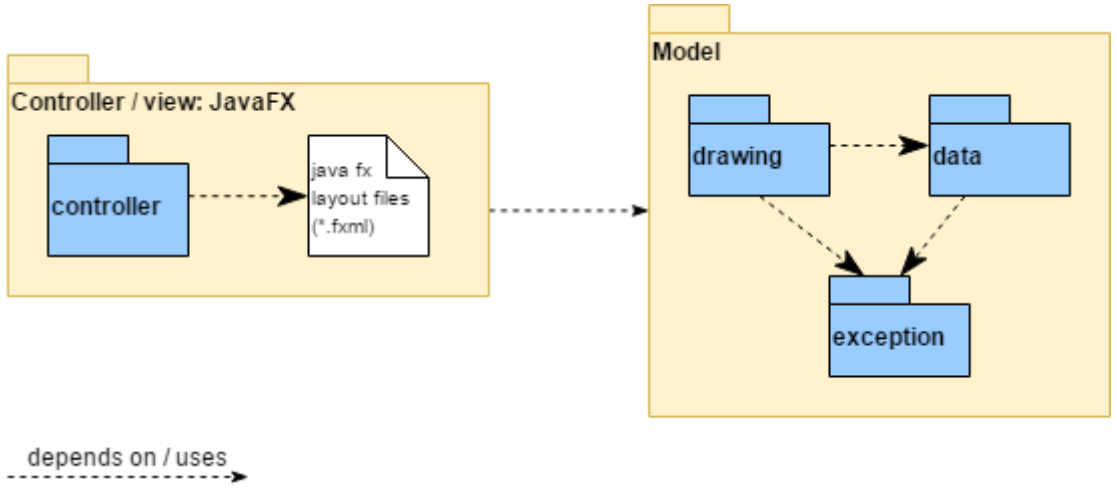
Software architecture views

There are several perspectives on our software, and each of them comes with a particular abstraction, an architecture, which describes the product from that perspective. We will be looking from four of these perspectives: subsystem decomposition, hardware/software mapping, persistent data management and concurrency.

subsystem decomposition

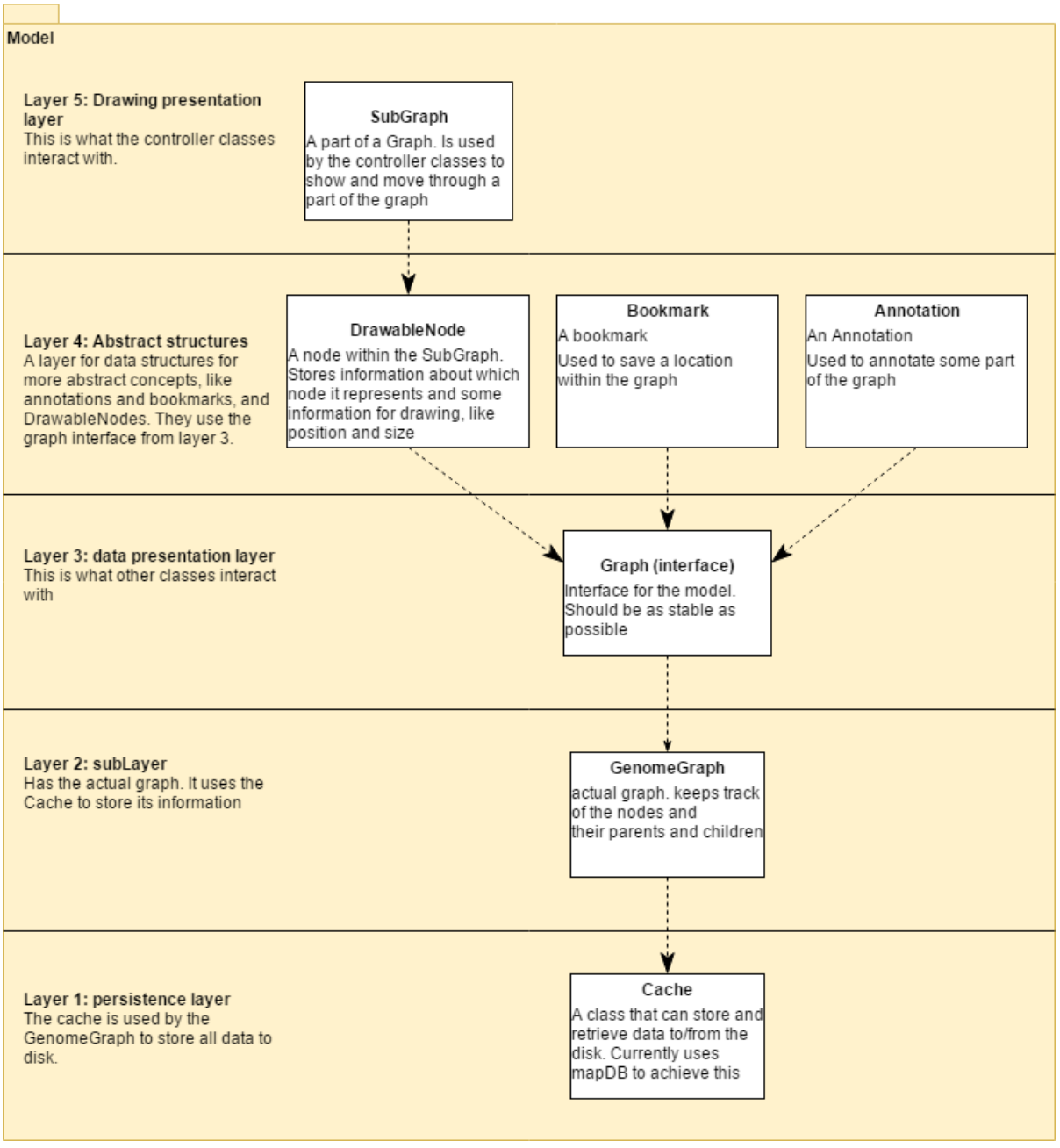
The subsystem decomposition perspective deals with the internal design of the program. It tries to divide the program into multiple more or less independent modules which should keep interaction between them to a minimum (loose coupling), while interaction within the modules should be very high (high coherence).

To achieve this, we will be using the model-view-controller design pattern (MVC), but controller and view are mixed because we use JavaFX and it doesn't really give a good way to keep controller and view completely separate.
For JavaFX, we have `*.fxml` files for laying out the static part of the program, but the dynamic parts of our program (the graph) is unfortunately mixed with the controller.
The model contains packages drawing, data and exception. Drawing are the data structures used for drawing (Subgraph that can be dynamically moved through the graph), data are structures that are only used for holding data (like the graph), and exception is a pack



We will also use a layered architecture in the model part to make it possible to easily swap out unused parts of the data structure without it being visible from higher layers.
As of writing this document, the graph used Segments to keep track of the nodes. But we will be removing them from the graph in favor of using just `int`s to represent the ids of the nodes. This will not affect the users of the graph, as they simply use the graph interface without knowing how the graph stores the nodes.

The graph is also responsible for saving all information to the cache. Users of the graph only need to close it in order to use this feature correctly, but they are otherwise unaware of the graph storing and retrieving data from disk.
Note that this is the inverse of the original planning, where the datamanager would use a graph datastructure to store everything, instead of the graph using the the cache (new name for datamanager).



hardware/software mapping

Our entire software is made to work on one computer. It is written in Java, which means that it is mostly the JVM that decides where and when our code will run (for example, there is no way to specify that two threads should run on different cores / cpu's). There is actually very

little we can do in terms of choosing what type of hardware we use, so we will not be looking into it.

Persistent data management

Our application may have several sources of (semi-)persistent data. First, we have the .gfa files. Storage of these is not the responsibility of our application: it only needs to find them in the file system and then open them.
Next, we will need to store bookmarks, recently opened files, caches (on disk storage to increase speed of reloading a file), and possibly configuration settings in the future. This is currently done as files in the same folder as the jar of the program, but we may be moving it to a folder in the same folder as the jar in order to make it easier to delete everything at once.

Concurrency

Threads

At all times, the instances of our program will be completely separate, which means that for the purposes of our program, there might as well be only one instance running. So there is no concurrency in the form of a distributed system or client / server setups. However, we will use concurrency within the program (in the form of threads) to provide a smoother user experience. We will use background threads to load data from disk while the user doesn't need it yet when it is determined that he may need it in the near future, so that it is already loaded and processed when the user does need it. We can also use this technique to quickly load a small part of the graph when opening files, so that there is some output as soon as possible (this increases the perceived performance), even though the file has not yet been loaded completely.

Further, we will definitely use concurrency for the GUI, to make it as responsive as possible, and to give the user the possibility to cancel any long operations.

Shared resources and deadlock

We will only have hardware as a shared resource. Managing the hardware is not our job, it is the job of the JVM, which probably delegates some or all of that responsibility to the OS.

Glossary

Perceived performance: When the program feels smooth it is not needed to be actually very fast.