Architecture design

Desoxyribonucleïnezuur, context project 2017

Toine Hartman - 4305655
Martijn van Meerten - 4387902
Iwan Hoogenboom - 4396634
Yannick Haveman - 4299078
Ivo Wilms - 4488466

3 May 2017

TODO: table of contents

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”). As the way we work (scrum) is by definition adaptable and favors change, this design is not set in stone: we may change it 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. TODO: add better intro

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

Of course, good code quality is desirable for any software project, but for our project it is an actual requirement: we get graded for it. 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

Even though the process is not really part of the design, it is definitely a goal: we are, again, graded for 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 (for us) 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 (“TDD”), 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 should be able to manage memory by using a space-efficient data structure and writing unnecessary parts of it to disk (or maybe throwing it away entirely and reparsing when needed), and loading it when needed / when it is determined that there is a high chance we will need it soon (as a background process, concurrency again).
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 search for / 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

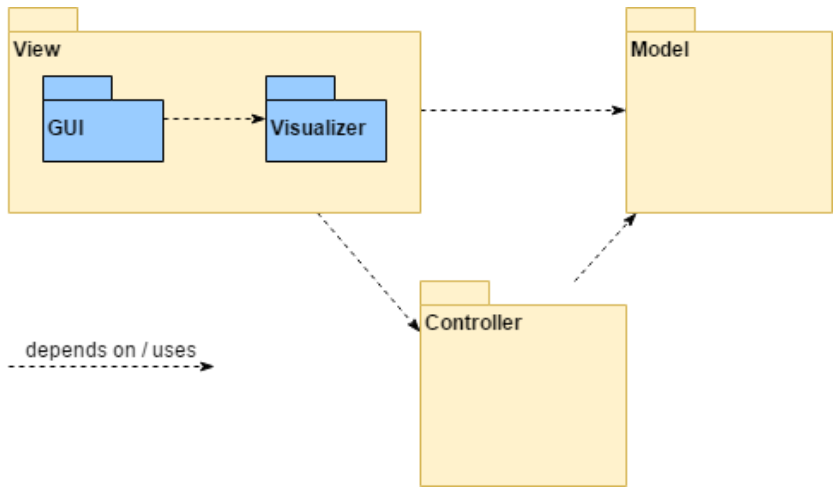
While we obviously value actual performance, there is no reason not to give the user the idea our program is quicker than it is. If users get the feeling they are waiting less, even if they actually are not waiting less, we decrease frustration, which increases the (apparent) usability of our program.

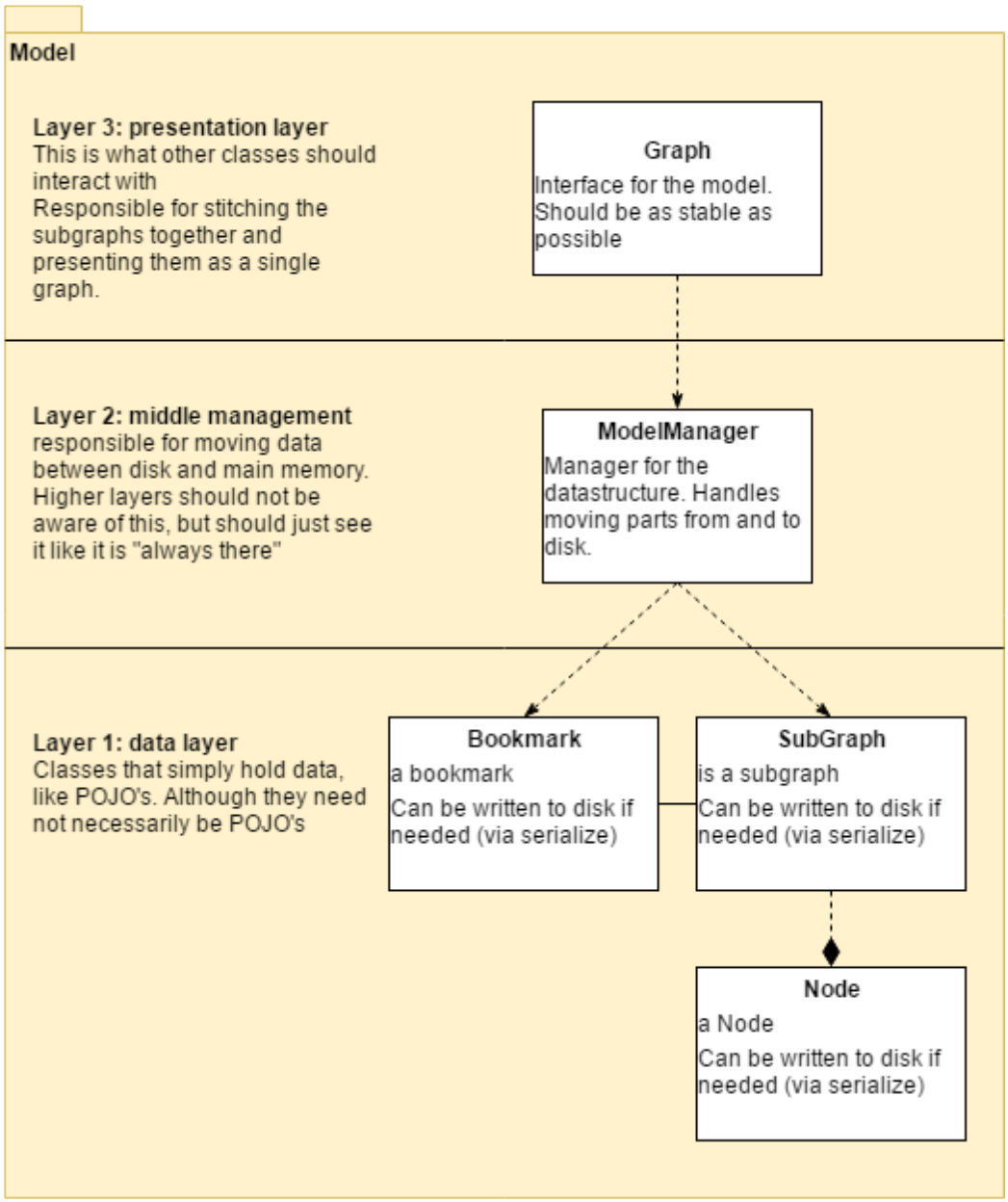
Software architecture views

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), with also 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.





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). So in conclusion, we just have to make multiple threads and that the JVM doesn't choose a hopelessly ineffective way of managing them. Perhaps we could help the odds by sacrificing a goat to the JVM gods.

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 and other file specific (meta-)data. This will most likely be in a different file with our own standard, but we could also save it to the .gfa file. We will need to research the possibilities and see whether there are any standards we could use. But we are currently not even close to implementing bookmarks, let alone saving, so we will look into it when we get to it.
We might also make some configuration settings, and it would of course be useful from a usability perspective to save these settings. We will probably use our own file somewhere that is loaded automatically on startup. Again, this is for the future, we first need to get the core features working.
Last, we might save some part of the file structure to disk to save memory. This is basically a temporary file, and we probably won't need to save it between runs at all (it is used as virtual memory), which means we can remove it when closing the program. We could also leave it there to speed up running the next time, but that would take up disk space, and we would need some way to check that it is the same file that is opened, i.e. that the cache is actually for the currently opened file. Yet again, we will evaluate these choices when they become relevant.

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. Rendering will also be done in a separate thread, to get separation of concerns and to keep the program easy to understand.

Shared resources and deadlock

We will have two kinds of shared resources, hardware and data structures. 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). Managing the data structures is our job. The rendering thread, the GUI thread and the background thread for efficiency will use these.

The background thread has the highest priority, since it can remove parts of the data structure by writing it to disk. We must either have some sort of “cancel write” message that cancels writing a part of the data structure back to disk, or make the data structure in such a way that a used part cannot be actually removed from memory until the thread using it is done. This is actually already implemented by the garbage collector.
If one of the threads requests a part of the data structure that is currently on disk, it can obviously not be used directly. The background thread (or yet another thread) will need to load it from disk before it can be used.

The rendering and GUI threads remain. There are two ways to prevent deadlocks between them:

- lock the data structure while rendering is in progress. This may hurt responsiveness, but is pretty easy to implement.
- Don’t lock the data structure while rendering is in progress. This will give ugly effects if the data structure is modified during rendering (and it may even create infinite loops if we are really unlucky), and we will need to ensure that the rendering process is called again afterward to fix any mistakes. However, the GUI thread doesn’t have to wait and the program becomes more responsive.

We have yet to decide the best way to solve these issues

Glossary

TODO