Emergent Architecture Design

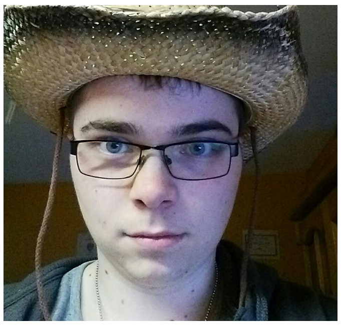
Context project - Programming life

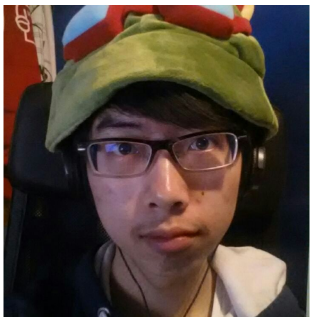
June 24th 2015

Justin

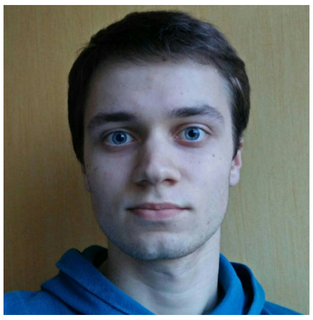
Marissa

Mark





Chak Shun
Scrum Master



Maarten
Product Owner

Team PL1

Chak Shun Yu	csyu	4302567
Justin van der Hout	jrtevanderhout	4319982
Mark Pasterkamp	mpasterkamp	4281500
Marissa van der Wel	mmvanderwel	4323602
Maarten Flikkema	mhflikkema	4306538

Table of contents

1. Introduction

This document will flesh out the system that is built during the Context Project ‘Programming Life’ by our team. We will explain the architecture of the system in terms of high level components of the system.

1.1. Design goals

We strive to maintain the following design goals throughout the project:

Availability

There will always be a working version of the product, so that the client can use it and give feedback. This feedback can be used to adjust the product.

Maintainability

To allow later additions and changes to the system, the code will be well documented, closed for modification and open for extension. To provide the ability to adapt the program in a later stage, we aim to work following the SOLID principles[1].

Performance

When data is imported for the first time, it is converted into a format that is faster to read and saved into a file. This way, the following times the data is imported it can happen a lot faster. Also, it is useful to store derived data like semantic zoom level graphs.

Scalability

The system should be able to handle DNA sequences millions of base pairs long. This means the system needs to be efficient so that it won’t take too long to process once it gets scaled up.

Securability

Some of the data that is worked with is possibly confidential. Therefore our program will be runnable offline on a local computer, without any need of internet. This way we exclude the possible chance of data being leaked to the outside world.

Useability

Biologists should be able to work with the application in a recognisable way. There should be no need to “study” to be able to use it. Functionalities visible to the user must always be useful at the time it is visible.

Code quality

To ensure the quality of our code, we use analysis tools to statically analyse the code. These tools include CheckStyle, PMD, FindBugs and Cobertura.

We also want to make sure that the master branch is always operational. To achieve this we make use of git branching and pull-requests. Every task is worked upon at a specific branch that is separate from the master branch. When the work on a branch is finished, a pull-request is initiated on the branch and the other team members will evaluate the code.

[1] https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

Evaluation of the code is based on comments/documentation, automatic/manual testing, completeness and whether it can be built by Maven.

2. Software architecture views

2.1 Overarching architecture

The architecture of the system is best described as a blackboard architecture. All data is stored in one class called the DGraph (Data Graph). The DGraph consists of instances of DNode and DEdge. Also, gene annotation and known mutations are stored in the DGraph via an instance of ReferenceGeneStorage. Only the phylogenetic tree is stored one level higher in the hierarchy, namely in the graph panel containing the data graph.

We chose for this architecture because this allows us to keep all data storage in one place. The amount of data we get is enormous and so much that an object orientated language cannot handle all this data if loaded all at one. Because all the data is now in one place, extending our data storage to use a graph database becomes much easier.

Design choices

Graph representation and User Interface

We decided to use the GraphStream[2] library for the graph representation, since it seemed to provide a good basis for graph representation. Using this library we quickly had a visual representation of the provided data. After deciding to use GraphStream, we found it would be best to use the Swing library for the complete user interface, because this would make GraphStream directly compatible with the entire graphical application.

Later into the project we found out that using GraphStream as a basis might not have been the best choice. If we had written our own way of representing a graph, it would have been more open to extension, it would have been easier to define its behaviour, and we would have been able to use more visual aspects, as prefered by the customer. When we discovered this however, we were already quite far along with the program, and felt it would not be optimal to switch.

Score Multiplier

We decided to use the singleton pattern for the ScoreMultiplier class. This class keeps track of all the multipliers that influence the score for mutations. We felt it would be best to make it a singleton since there should be only one at a time, and it will be used by every single Mutation.

Mutations

For the Mutations we decided to make an abstract class that would be extended by all different kinds of mutations. We did this because certain methods would be similar regardless of type, and this would mean we would have to use duplicate code in case of an interface. We decided to not use a factory pattern for the creation of mutations, since every

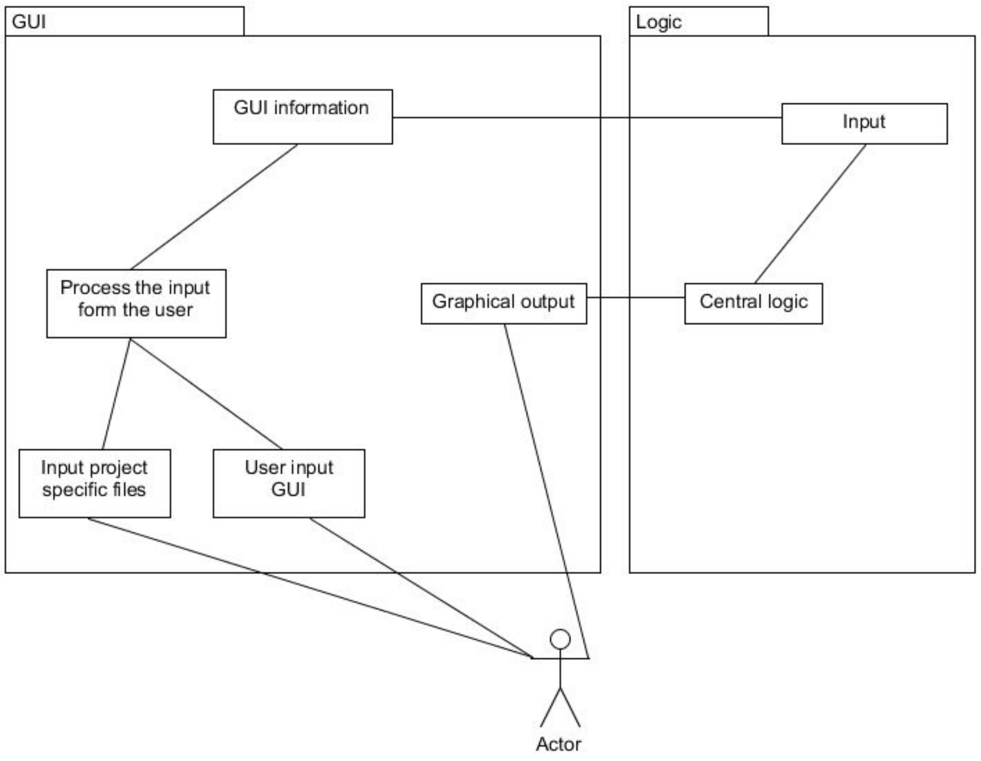
[2] http://graphstream-project.org

mutation needs different parameters, and this would mean every single mutation would need its own factory. This did not seem optimal and so we decided not to use this pattern.

2.2 Sub-system decomposition

At the highest level of overview, as can be seen by the figure below, we can describe the system as 2 parts: the graphical user interface and the logic of the program. The GUI is the visualising part of the program and lets the user interact with it. The logic is the component which executes computations on the data, such as the collapsing of nodes for different zoom levels and genome highlighting.

The user interacts with the GUI, for instance, by loading a file or choosing which zoom level he wants to see. When this happens, the input is processed and the information is sent to the respective logic part of the system. The logic then executes computations on the processed input and the result of the computations is then sent back to the user as graphical out in the GUI.



When we take a look at the GUI, we can divide it into three major components:

1. The graph panel;
2. the phylotree panel;
3. the option panel.

The graph panel is the panel in our GUI in which the graph is shown. When you highlight a genome or go to another zoom level, you can see these changes in the graph panel. When you select a node in the graph panel, you will see all the information about this node (like how many sources go through this node).

The phylotree panel is the panel in our GUI which shows the phylogenetic tree. The phylogenetic tree is the tree that shows how bacteria are related to each other. When you left click on a node in the phylogenetic tree, all it's children will be selected and you can see its selected genomes now being highlighted in the graph panel. When you right click on a node in the phylotree panel, it and all its children will be collapsed.

The option panel is the panel in our GUI in which you can set the options. In the option panel, you are able to set the different options in our program like highlighting genomes or moving sliders which are used to indicate which mutation you find more interesting for semantic zooming.

Our GUI is backed by our data model, which is called DGraph (Data Graph). The DGraph consists of DNodes and DEdges. The DGraph is the part of our program in which all our information is stored. Which is why most other classes have access to the DGraph to get their information.

The last part of our system consists of the logics part. This part of our system is shared across multiple classes, but mostly the calculation of the zoom levels and the mutations..

2.3 Hardware / Software Mapping

There are no strict hardware requirements for our program. It is however restricted by available memory when loading large graphs. Other parts of the program such as the phylogenetic tree, gene annotation and known mutations are not restricted by this as they are not too large in size.

A fast processor is greatly recommended, as reading the input files and collapsing and placing nodes take many computations.

2.4 Persistent Data Management

The persistent data that our program mainly uses is the initial uncollapsed graph. We keep only this data in memory because it would be less efficient to keep all zoom levels in memory, and we did not have capabilities to put it on disk yet. By having the highest zoom level in memory, we can use that to compute every zoom level. The metadata will also stay in memory once it has been imported. It does not take up a lot of memory, and the user would only load in the metadata if they plan on using it.

Our initial plan was to make use of a database from which we would only load small parts of the graph. This would reduce the computation time of importing data that has been imported before. However, due to time constraints we were not able to implement this. Instead we now always read the graph from the provided text files, so we do not save computation time of the placement of nodes and we keep the entire graph in memory.

2.5 Concurrency

We chose to only utilise concurrency while reading and loading the graph into memory. We wanted to keep the GUI responsive while loading a graph, so we used a SwingWorker for that. Later on we realised our application wouldn't have much functionality without the loaded graph, so we added a modal progress dialog. The use of the SwingWorker makes sure the application does not become unresponsive.

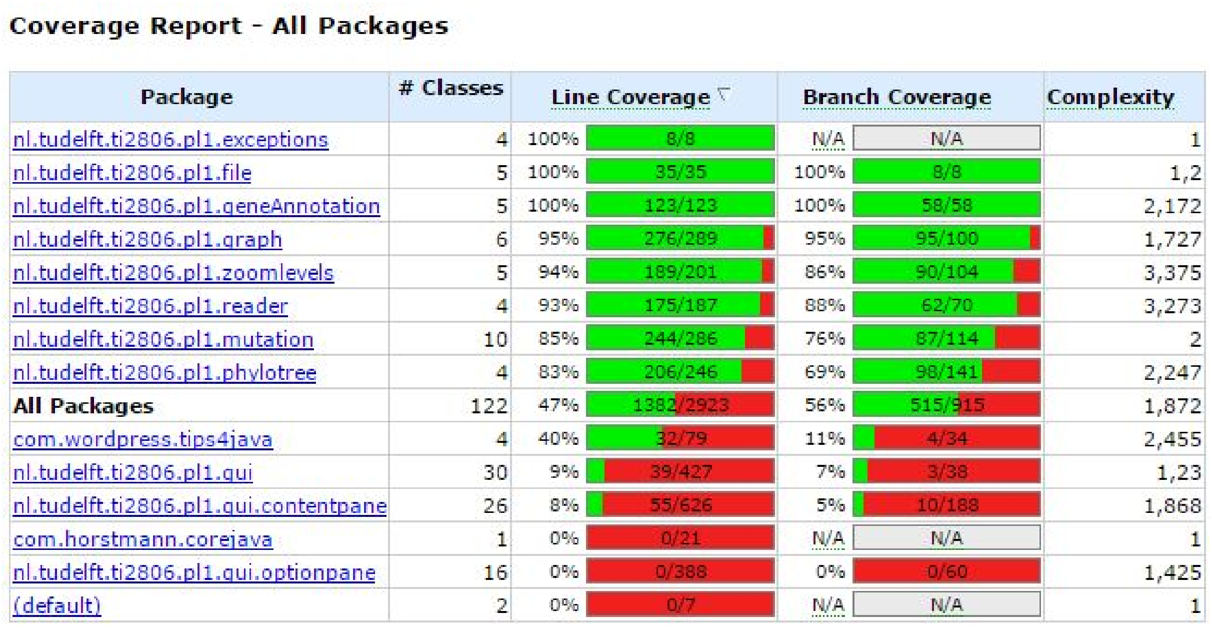
Appendices

A. Testing

The source code of DNApp can be split into three parts when considering testing:

1. The part of the code base we wrote which is well tested by automated JUnit tests,
2. The part of the code base we wrote creating the graphical user interface, which is well tested manually, and
3. Two open-source-like classes we found on the internet which we did not test ourselves.

In the image below, you can see the distinction of these three categories quite well in terms of code coverage.



Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
nl.tudelft.ti2806.pl1.exceptions	4	100%	8/8	N/A	N/A	1
nl.tudelft.ti2806.pl1.file	5	100%	35/35	100%	8/8	1,2
nl.tudelft.ti2806.pl1.geneAnnotation	5	100%	123/123	100%	58/58	2,172
nl.tudelft.ti2806.pl1.graph	6	95%	276/289	95%	95/100	1,727
nl.tudelft.ti2806.pl1.zoomlevels	5	94%	189/201	86%	90/104	3,375
nl.tudelft.ti2806.pl1.reader	4	93%	175/187	88%	62/70	3,273
nl.tudelft.ti2806.pl1.mutation	10	85%	244/286	76%	87/114	2
nl.tudelft.ti2806.pl1.phylotree	4	83%	206/246	69%	98/141	2,247
All Packages	122	47%	1382/2923	56%	515/915	1,872
com.wordpress.tips4java	4	40%	32/79	11%	4/34	2,455
nl.tudelft.ti2806.pl1.gui	30	9%	39/427	7%	3/38	1,23
nl.tudelft.ti2806.pl1.gui.contentpane	26	8%	55/626	5%	10/188	1,868
com.horstmann.corejava	1	0%	0/21	N/A	N/A	1
nl.tudelft.ti2806.pl1.gui.optionpane	16	0%	0/388	0%	0/60	1,425
(default)	2	0%	0/7	N/A	N/A	1

User Interface

Since the coverage of the GUI packages is not well represented by this measurement, we will elaborate a bit on how we did test the gui.

We did not automatically test the GUI by using JUnit or another framework. This is because it is relatively difficult and time consuming to write JUnit tests for the GUI. Certain functionalities of the system will only be accessed when using a specific sequence of GUI events. Depending on the length of this sequence, the effort to set up a specific functionality can increase exponentially. To do this manually for everything in the GUI would be very time consuming. It would also not be worth the effort because the GUI is only a link between the user and the actual functionalities underneath it, which are already tested using JUnit tests. This means when testing the GUI we would only be testing whether certain input on the GUI would result into certain events, which is not very rewarding.

Instead of JUnit tests, we resorted to manual testing the graphical user interface. The way we did this was by frequently building and running the program while developing new features. We would then try using the new implemented feature and briefly go through the old functionalities for the sake of regression testing. Additionally, we made a policy that for every pull request we made on Github at least two other team members, that were not involved in the development of that particular feature, would pull the new code to their machine and test the new feature. By practicing this, we ensured that at least 60% of our team would have thoroughly tested the interaction of the GUI with the new feature and thus also the GUI itself.

Given we had more time, we could have tried to use a external tool to help with testing the GUI. We could have used Marathon to test it in general or perhaps UISpec4J/FEST, which are specifically designed to be used for the Swing library that we use. These tools we could invoke certain methods, (click) events, triggers or mimic certain behaviour and test whether the GUI would work as intended when used by a user.

B. Glossary

Branch - A branch is a separate environment of the master branch. Its purpose is for the contributors of the project to make modifications to the code, which can then be evaluated before they are added to the master branch.

CheckStyle - A tool that finds Java programming rule violations.

Cobertura - A tool that measures to what extend the code is covered by automated tests.

FindBugs - A tool that helps finding bugs in the code.

GUI - Graphical User Interface.

Master (branch) - The main code. This code should always work and be of the highest possible quality.

Maven - A software project management tool.

Metadata - The data representing the phylogenetic tree, gene annotation and known mutations.

Multithreading - The ability of a program or an operating system to handle multiple processes simultaneously.

PMD - A tool that finds general flaws in the code.

Pull-request - When the work on a branch is finished, a pull-request is initiated on the branch. A pull-request can then be accepted by the other team members and the branch' modifications are added to the master branch.

Semantic zooming - A variant of regular zooming in which zooming does not only bring an object on the screen virtually "closer" to the user, but also reveals more or less details depending in lower respectively higher zoom levels. Example: Google Maps (shows country names zoomed out, but street names zoomed in).