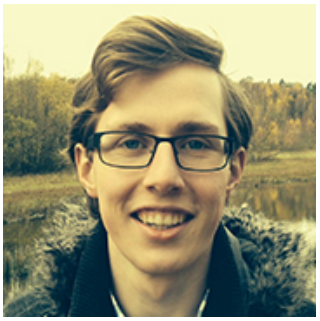
Geex - Emerging Architecture Design

Contextproject Programming Life - Group 2
TU Delft



Gerben Oolbekkink

4223896

Jasper Boot

1516272

Jasper Nieuwdorp

4215796

Jim Hommes

4306090

René Vennik

4102959

June 24, 2015

Contents

1. Introduction

This document provides guidance alongside our project build in the context project programming life. This document will be updated throughout the project and will explain all the systems and their subsystems to give an overview of the software engineering aspects. For an insight in the current status of the project also visit http://geex.hup.blue/, http://sonar.hup.blue/ and https://github.com/Vennik/contextproject/wiki.

1.1. Design goals

Availability

Always have a working product at master. The client should be able to download a build at any time from the release page to the affect that the client can test it and propose desired changes.

Interactivity

We want an efficient to use, quickly responding program to visualise multiple genome strands and see their underlying relations and differences (mutations) in an interactive way.

Scalability

We would like to be able compare around 500 strands of DNA on a single workstation. But 6000 strains could also be useful in the future; this is a matter of scaling the hardware with the data.

Modifiability

The program should be easily modifiable since extra filter could be added after we are done with it.

Reusability

The components of the program should be made with re-usability in mind. Dr. T. Abeel indicated that he wants to use the best parts of each program, so it should not be hard to reuse parts of the project.

2. Software architecture views

The system pre-loads data stored in textual files. Text files for the edges and nodes of the DNA strands and Newick tree format for the phylogenetic tree. These files are stored locally. There will be a controller to select different options and a viewer that will display the graphs. There is only one interface to this program. For known design flaws please refer to the appendix (A).

2.1. Overarching architecture

The core of the geex application is based on an MVC architecture. There are different controllers, each with their own view. The Controllers are defined in the package nl.tudelft.context.controller in geex-core. Views are defined as .fxml files in the resources folder of geex-core. Models are found in the geex-models module, along with their parsers.

This architecture is probably the most suitable for any application using JavaFx and backed by a data set. Most of the view and interaction is handled using JavaFx, this makes it easy to split the controllers from the views. There are some cases where creating fxml files was deemed too complex, here the choice was made to inject controls into the view using a controller.

In the code the border between models and the controllers is very clear because they both are in different maven modules, this makes sure that any communication between these two modules only happens from controller to models and not the other way around. It is impossible to reference a controller from the model and still have a passing build. Controllers and views are split by language, .fxml files are views and .java files in the controller package are controllers.

2.2. Sub-systems

There are currently six modules in the geex project. Dependencies between modules must be unidirectional, preventing cycles between modules.

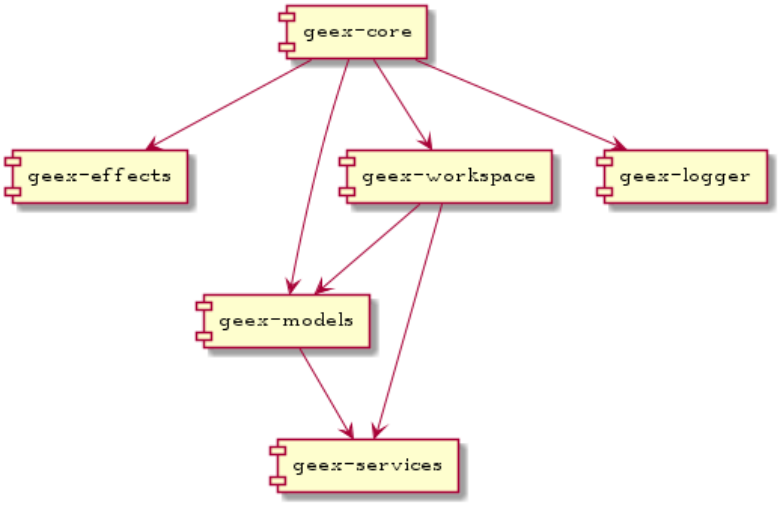


Figure 1: Modules in Geex taken from Maven site

2.2.1. Geex Models

Geex models is the module containing all models and parsers. It contains models for the Newick, graph, annotation data and resistance data. Each type of model has it's own parser.

Graph Graph data is loaded from the .node.graph and .edge.graph file in the workspace. JGraphT is used for saving the data to objects.

There is also a filter package in graph. All of these filters are implementations of `StackGraphFilter`. Filters are registered in `GraphFilter` in Geex Core. Filters are created using reflection, it is important that the constructor of a graph filter doesn't have anything that can fail.

The filter package first used an incomplete decorator pattern. This was removed because filters don't actually add anything to a graph, they just do some operations on the graph. Using a decorator pattern is also too complex, because `StackGraph` extends `DefaultWeightedGraph` from JGraphT, which has a lot of functionality.

Newick For loading the Newick files a modified version of the Treejuxtaposer (http://olduvai.sourceforge.net/tj/), hosted at https://github.com/qurben/libnewicktree is used for parsing the files. Newick files also contain position data, so this is also saved in the model.

Annotation There are two types of annotations, drug resistance data and annotation data. Both are loaded in the annotation package.

The drug resistance data is a custom file format, for which we created our own parser. The following represents a single line in the drug resistance data.

```
geneName:typeOfMutation,change,filter,genomePosition   drugName
```

The annotation data is a tab separated file. We also created our own parser for this.

```
seqId  source  type  start  end  score  strand  phase  attributes
```

2.2.2. Geex Core

Geex core is the main entry point of the application. It is responsible for showing and building the views and routing the views. This module also contains the App class that runs the entire application.

Controller The controller package contains controllers for different views. Controllers extend `AbstractController`, this way a .fxml file can be bound to this controller.

Drawable The drawable package contains drawable versions of the newick and the graph.

2.2.3. Geex Services

Geex services contains helper classes for loading data asynchronously. Any class implementing Loadable can be loaded in a separate thread by `LoadService`. This class can load any file using a parser which extends `Loadable`. The parsers from Geex Models are loaded using `LoadService` in Geex Services.

A `Loadable` class is instantiated in a different thread and given a set of files (zero or more) to use as base for processing data. The data can be returned using the `load() : T` function in a loadable. A loadable must also stop as soon as the outcome of `isCancelled()` is true. This means that in each loop this function must be checked.

2.2.4. Geex Workspace

Geex workspace is the module which is responsible for managing the current workspace. It also includes a small database class for saving previous workspaces. The workspace provides `ReadOnlyObjectProperty` objects for each type of model it provides. This way any controller can bind to the change of this property and will be notified when the workspace is loaded.

2.2.5. Geex Effects

Geex effects is a helper module which is currently only used for creating a zoom effect on nodes in the graph. This package should also be used when new effects are created. Effects should be independent of their context and just work on any JavaFx object or specify interfaces they will work on.

2.2.6. Geex Logger

Geex logger is a small module which is responsible for logging data. It includes a Log class which is able to log messages to several different Loggers, there are currently two loggers. The first logger is the StdOutLogger, which is part of Geex Logger, it writes data to System.out when the software is run with the -Ddebug=true jvm option. The second logger is MessageController, which displays info and warning messages on the bottom of the screen.

New loggers can be created by implementing the `Logger` class. These loggers can be registered to the main logger using `Log.instance().addLogger(myLogger)`. There are currently three log levels, debug (1), info (2) and warning (3). A logger also has a log level. Messages are sent to any logger having a log level which is smaller than the log level of the message. This way a Logger can choose to only show info and warnings.

2.3. Hardware / Software Mapping

The software should run on one single workstation, of which the specifications are unknown at this moment. We suspect an Unix based operating system with at least 2GB of RAM and a current processor. We will use Java so it can run on any operating system. A single researcher will use a single workstation so data persistency should not be an issue, since files are loaded only once and not edited.

2.4. Persistent Data Management

There is persistent data management in the geex-workspace module. There a database-connector is used to save the previous workspaces to a database file. The sql connector used is SqlJet. SqlJet is an Sqlite implementation in pure Java with a very small footprint.

2.5. Concurrency

All models are loaded in a background worker. The geex-services module is responsible for this. Background workers are created as JavaFx services. Their valueProperty is returned to any other class using the value of the worker.

Using these properties we can make other classes independent of the background worker. Classes depending on the data which is to be loaded or is already loaded get a ReadOnlyObjectProperty, they can then listen to changes of this property.

Currently asynchronous tasks are only used for reading data into the system. There are no cases where there could be a race-condition or deadlock. Different tasks don't depend on each other.

3. Glossary

- Genome: the haploid set of chromosomes in an organism
- Strands: a reading of the genome expressed as a list of A, G, C and T's.
- Mutation: an alteration of the DNA, this can be deletion, insertion, translocation & inversion
- Newick tree format: a file format to store trees in a textual representation.
- Phylogenetic tree: diagram showing the inferred evolutionary relationships among species.

Appendices

A. Known Design flaws

MenuController Currently the class `MenuController` is a data class because other classes use the data in this class. What happens is that the objects in `MenuController` are accessed by other classes (via accessor methods) and they then invoke methods. For example when `MainController` wants to switch a view it will use `getToggleOverlay()` to invoke `fire()` on the accessed `toggleOverlay`. This is not desirable because it will access the whole object when it only needs to invoke a single method. This could be fixed by implementing the command pattern. This would mean that `MenuController` would get an command interface that has access to `MenuController` while shielding it from the other classes. Other classes then can use this command interface to invoke commands on the `MenuController`.

Logger The logger is currently not complete, this causes errors in SonarQube. SonarQube wants a logger as a private variable and the possibility to create a logger for a specific class.

```
private static final Logger LOGGER = LoggerFactory.getLogger(Foo.class);
```

One needed addition to the logger would be the specification of a class. This way it can be logged in what class an error occurred.

B. Testing

If you look at our test coverage you can see some spikes in branch coverage and line coverage. The most remarkable class here is `App`, which has a coverage of 0%. This is an example of a class that can not be tested, since it is the main start of the application.

Besides `App`, in some classes we have complex operations with the mouse, for example drag-and-drop operations. Some of these operations have been tested with `MouseEvent`s, but others are harder to test. Simply simulating the MouseEvent and checking the result afterwards is a process that is hard to test. That is also the reason why we did not do this.

Another reason not to test certain features is because their result is visual. While we can see ourselves whether this works or not, writing tests for visual features is not possible. An example of this is the Shift feature of a drawable node, that occurs when an In-Del is detected. This also lowers the overall branch coverage.

Finally, there are also some edge cases in the project that have not been tested. While we should strive to test every line and every branch, given our time frame that is simply not doable. In this project we were more focused on producing the code rather than testing every possible branch. That combined with our time frame is reason enough for the coverage we missed.