PHILIP MACHANICK

# GENESIS 2 IMPLEMENTATION DOCUMENTATION VERSION 1.1.1

H3ABIONET

PUBLISHED BY H3ABIONET

FUNDED BY THE NIH, GRANT 5U24HG006941-10.

https://www.h3abionet.org

p.machanick@gmail.com



*First printing, May 2024*

# *Contents*

# *List of Figures*

# *Introduction*

This documentation describes experiences with bringing the Genesis 2 project to the point of being usable. Genesis 2 creates, displays and allows output of charts showing population principal component analysis (PCA) and admixture charts. Genesis 2 is based on an early Gensis tool that was implemented in Java using the Standard Widget Toolkit (SWT)[1]. The original genesis program was maintained up to August 2021[2]. Because of limitations of SWT, the project was re-implemented Using JavaFX, starting in 2019[3], with initial development by Henry Wandera.

Unfortunately, JavaFX was removed from the standard Java distribution as of JDK 11[4] in 2018. Consequently, it is no longer possible to rely on the build-once-deploy-everywhere promise of Java if JavaFX is used. Since the project was well advanced when I took it over, I stuck with JavaFX. The strategy I adopted was to make one build for each of Macs running the Intel architecture, Macs running the newer Arm architecture, Windows on Intel and Ubuntu on Intel. I also make a generic version that can be run by attaching to local JavaFX libraries.

JavaFX is designed to implement the model-view-controller (MVC)[5] design pattern in which user interface implementation is separated from data representation by a Controller layer. The architecture starts from designing interface elements in FXML[6], ideally using an interface builder. The interface builder could generate stub code for the Controller. This is how Scene Builder[7], a free interface builder tool (designed to work with FXML and, like JavaFX, no longer maintained by Oracle[8]), works.

In order to implement the MVC pattern correctly, there should be no interface-dependent coding in the Model layer. Anything used to represent on-screen drawing and manipulation should be translated to an interface-independent representation. Before anything is saved, the interface-dependent representation (View abstractions) should be converted to the Model. Restoring from saved should do the reverse: convert the Model representation to the View representation. Only

[1] https://www.bioinf.wits.ac.za/software/genesis/

[2] https://github.com/shaze/genesis

[3] https://github.com/h3abionet/genesis2

[4] https://www.oracle.com/docs/tech/java/javaclientroadmapupdate2018mar.pdf

[5] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988

[6] A user interface markup language based on XML, introduced by Oracle when JavaFX was part of the standard Java distribution: https://docs.oracle.com/javase/8/javafx/fxml-tutorial/why_use_fxml.htm.

[7] https://gluonhq.com/products/scene-builder/

[8] https://gluonhq.com/products/scene-builder/

the Controller layer should see both.

With those design considerations in mind, I have a starting point for a more robust implementation but the focus in this stage of the project was ensuring that the major features work so the program is usable. A big downside of this weak design is that any new version that cleanly separates View from Model will result in incompatible saved file formats as the program relies on serialising classes to save and relatively minor changes can break serialization[9].

[9] https://docs.oracle.com/javase/6/docs/platform/serialization/spec/version.html

In the remainder of this documentation, I start by describing the development tool chain, then outline lessons from understanding JavaFX, lessons from trying to make the existing code work and ideas for improved architecture based on those lessons. I end with a list of improvements that may not be very hard with more time.

# *Development Environment*

The main tool used is Apache NetBeans 18[10]. There are later versions but I decided to stick with the one I started with, since it is stable. Changes in more recent versions are unlikely to break development, Allied to this is the interface tool, JavaFX Scene Builder[11] (version 21.0.0 – again, updates should work but I stuck with a stable version).

NetBeans can be configured with a number of build engines; I use Apache Maven[12], which integrates into NetBeans using plugins. The plugins are named in the `pom.xml` file, which sets up dependences etc., much as a `makefile` does in a traditional Unix build. These dependences including required libraries should be resolved automatically when starting a build on a new platform.

The underlying Java engine I use is JDK 21[13]; I also use OpenJDK[14] and the two are interchangeable. The Java promise of "build once run anywhere" was always a bit tenuous [15] but the decision of current owners Oracle to remove JavaFX from the standard distribution means that a JavaFX application definitely cannot be built to run on any platform on which Java is installed.

The version of JavaFX that I use is 21.0.3 – at time of writing the latest version is 23[16]. I only upgraded from 21.0 to fix a small problem (a warning message that did not have any effect on functionality).

My main development platform is an M1 Max MacBook Pro, running macOS Sonoma 14 (14.4.1 when last built). I test and create releases on the following platforms:

- macOS Sonoma 14.4.1 – ARM 64-bit architecture
- macOS Monterey 12.7.4 – Intel 64-bit architecture
- Ubuntu 22.04.4 LTS – Intel 64-bit architecture
- Windows 11 – Intel 64-bit architecture

On a Mac or Linux, setup is similar. Check you have Java installed by running `java --version` on the command line; the version num-

[10] https://netbeans.apache.org/front/main/download/nb18/

[11] https://gluonhq.com/products/scene-builder/

[12] https://maven.apache.org/

[13] https://www.oracle.com/java/technologies/javase/21-relnote-issues.html

[14] https://openjdk.org/projects/jdk/21/

[15] Paul Tyma. Why are we using Java again? *Comm. of the ACM*, 41(6):38–42, 1998. DOI: http://doi.org/10.1145/276609.276617

[16] https://gluonhq.com/products/javafx/

ber should be at least 21.0. If not, download[17] and install a version (higher than 21 should also work).

Next, get a copy of the project files. For this, you need `git` installed; it is part of Apple's Xcode tools[18]. On Linux, if it is not installed, you will find instructions on the same page as the Mac install instructions. Go to a suitable directory then:

```
git clone -b Philip https://github.com/h3abionet/genesis2.git
```

Finally, get and install NetBeans[19]. A higher version than 18 ought to work but has not been tested. You should be able to open a project by navigating to the `genesys2` directory into which `git` downloaded the project.

For Windows 11 development, I use LXC to run Windows on the Ubuntu system. To do so requires a Windows 11 installer image; I follow the instructions in a tutorial[20]. This is reasonably robust except that LXC occasionally refuses to restart because it claims it has run out of storage (explained in more detail below). In that case I delete the instance[21] and start over. This is not as dire as it sounds as installing is fairly quick and as long as you do not make uncommitted changes on the Windows virtual instances, you do not lose any saved work.

When installing Windows 11 you do not need a product key; set yourself up as a local user. Once the Windows 11 virtual machine is set up:

- install JDK[22]
- install NetBeans
- install Git[23] – I use the Git Bash option, because that creates a terminal that allows other options
- using the Git Bash application, go to a convenient directory (convenience depends on how you use Windows: Desktop should be fine since this is not a generally used install) and get a version of the project as follows:
  `git clone -b Philip https://github.com/h3abionet/genesis2.git`
- now open NetBeans and open the project: navigate to the directory `genysis2` within your install location; it should find the `pom.xml` file, source files etc. and it should be possible to do a build

Run NetBeans; it may also be necessary to get JavaFX but the project setup should automate resolving dependences.

There is a bug in the configuration of storage size; the biggest I have been able to adjust this command to is 75GB; above that it insists that it must be a multiple of block size 16KB, which is clearly

[17] https://www.oracle.com/java/technologies/downloads/

[18] Or you can install it:https://www.atlassian.com/git/tutorials/install-git.

[19] https://netbeans.apache.org/front/main/download/

[20] https://ubuntu.com/tutorials/how-to-install-a-windows-11-vm-using-lxd

[21] Some ideas on how to do that here https://www.cyberciti.biz/faq/delete-container-with-lxc-lxd-command-on-linux/.

[22] https://docs.oracle.com/en/java/javase/11/install/installation-jdk-microsoft-windows-platforms.html

[23] https://git-scm.com/download/win

an error as 80GB is not accepted (nor is any multiple of 16KiB, i.e., powers of 2 rather than powers of 10 multipliers):

```
lxc config device override win11 root size=75GB
```

Consequently, LXC sometimes hangs and has to be reinstalled for allegedly lacking space. As a defence against this, I shut it down as soon as I have done a build and tests. Otherwise a reinstall from scratch is not too time consuming. And cheaper than buying a Windows machine.

To extract a build, a simple approach is to use `scp` to copy it to a real machine that supports `scp` – in my case, I use a Linux machine.

When you open the project in NetBeans the first view you have is as in figure 1. The panel on the left shows the file hierarchy. The most significant of these for the build is `pom.xml`, which contains dependences and in general drives the build. The line most frequently changed once a stable build is set up is the `version` line:

```
<artifactId>Genesis2JavaFX</artifactId>
<version>2.4b-CANDIDATE</version>
```

This line and the line above is built into the generated `.jar` files. One of the built files is the `artifactID` followed by the `version` and ends with `.jar`. This is the generic bulid that needs to be run on the command line, with the right version of JavaFX:

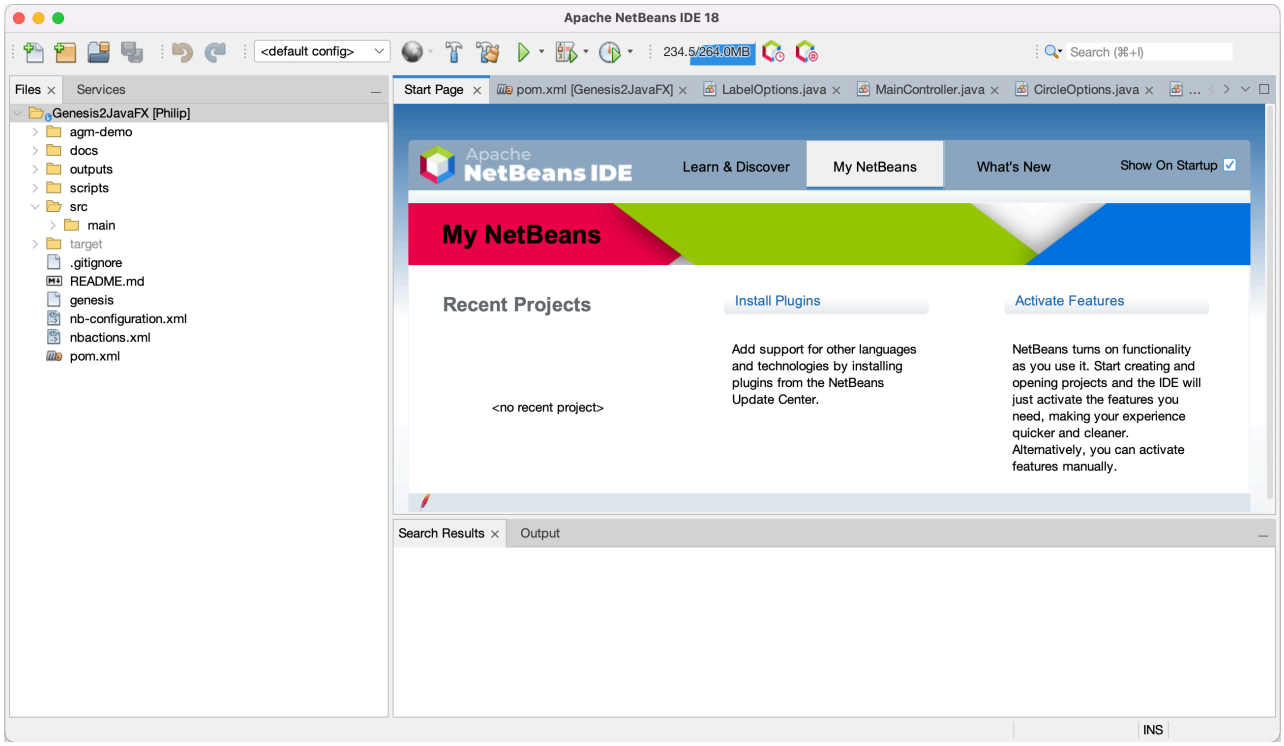


Figure 1: NetBeans after it opens

```
export JAVAFX=/usr/local/lib/JavaFX-21
export JARF=$HOME/Applications/Genesis2-2.4b-Generic.jar
java  -jar Genesis2-2.4b-ubuntuX64.jar
```

The version of the build should be corrected as should the path to the libraries and the `jar` file. If you download a build from a release on GitHub[24], the name of the `jar` file will be a bit different.

[24] https://github.com/h3abionet/genesis2/releases

The build generates a slew of messages similar to this:

```
Failed to build parent project for org.openjfx:javafx-base:jar:21.0.3
```

This probably should be fixed but does not result in any problem with the built artefacts. I have not yet found a fix for it.

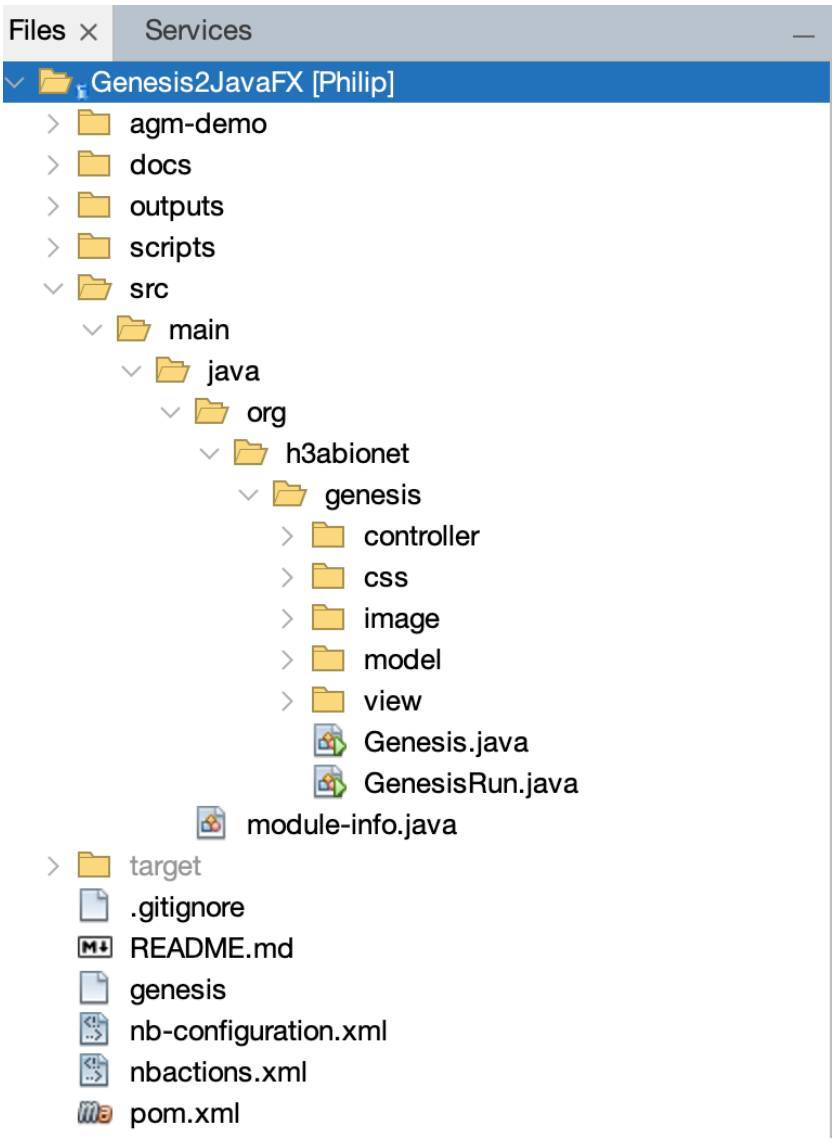


Figure 2: Source file organization viewed in NetBeans

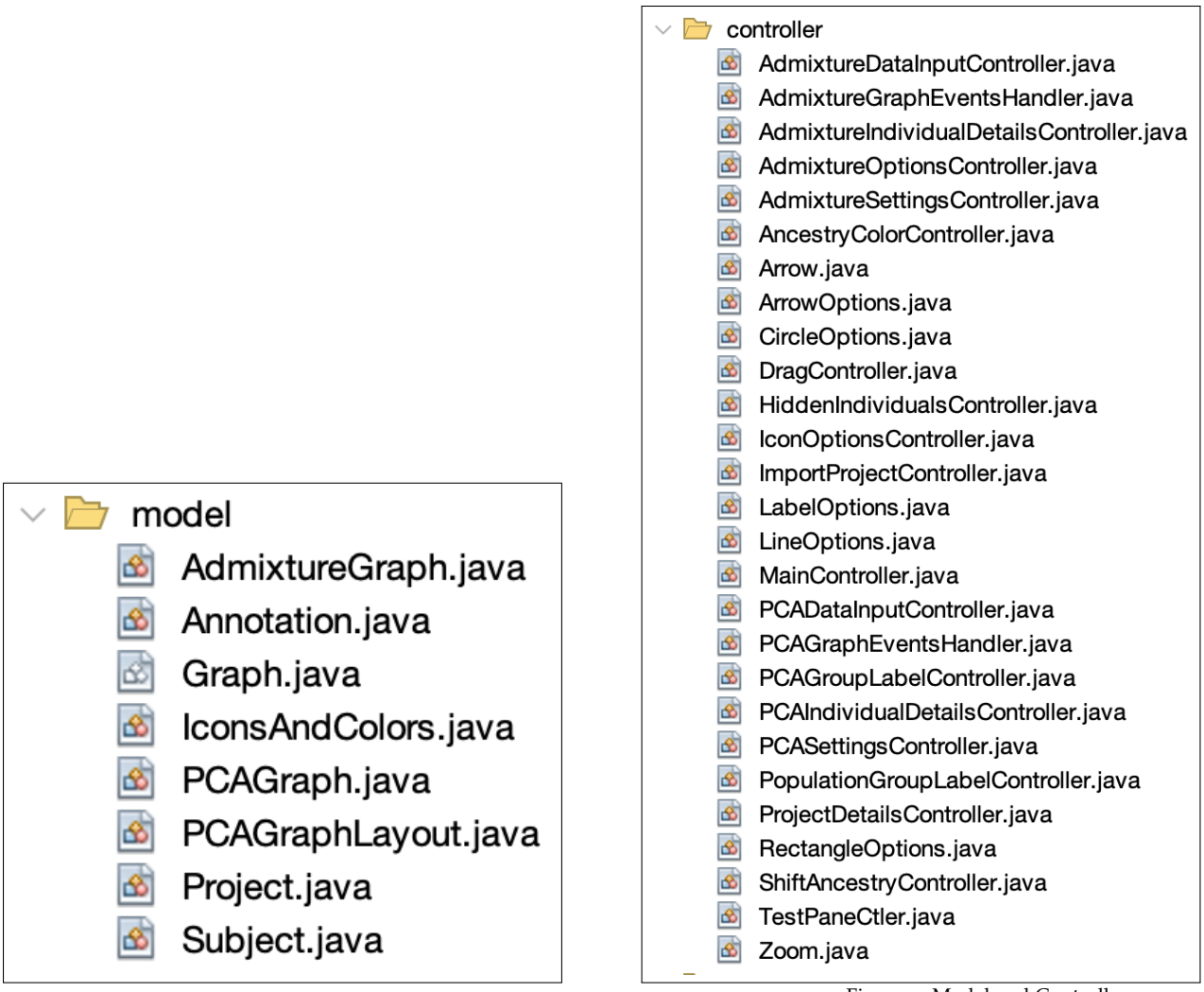


Figure 3: Model and Controller

If you click through the the `src` folder and others below it you will eventually reach the `genesis` folder, the contents of which are in figure 2.

If you open each of the sections under `genesis`, you can navigate to the source files. The organization is intended to fit the MVC model. The `view` folder only contains FXML files specifying graphics layouts and image files for elements of the user interface; the `css` folder also contains style details for the View.

Figures 3 and 4 show detail of the main modules. The View as pictured omits a long list of image files. The key thing to note about the View is that it is driven by FXML, which specifies layout, components and hooks into code in the Controller files.

In a proper MVC design, any additional user interface coding that specifies user interface elements (not using FXML or other declarative

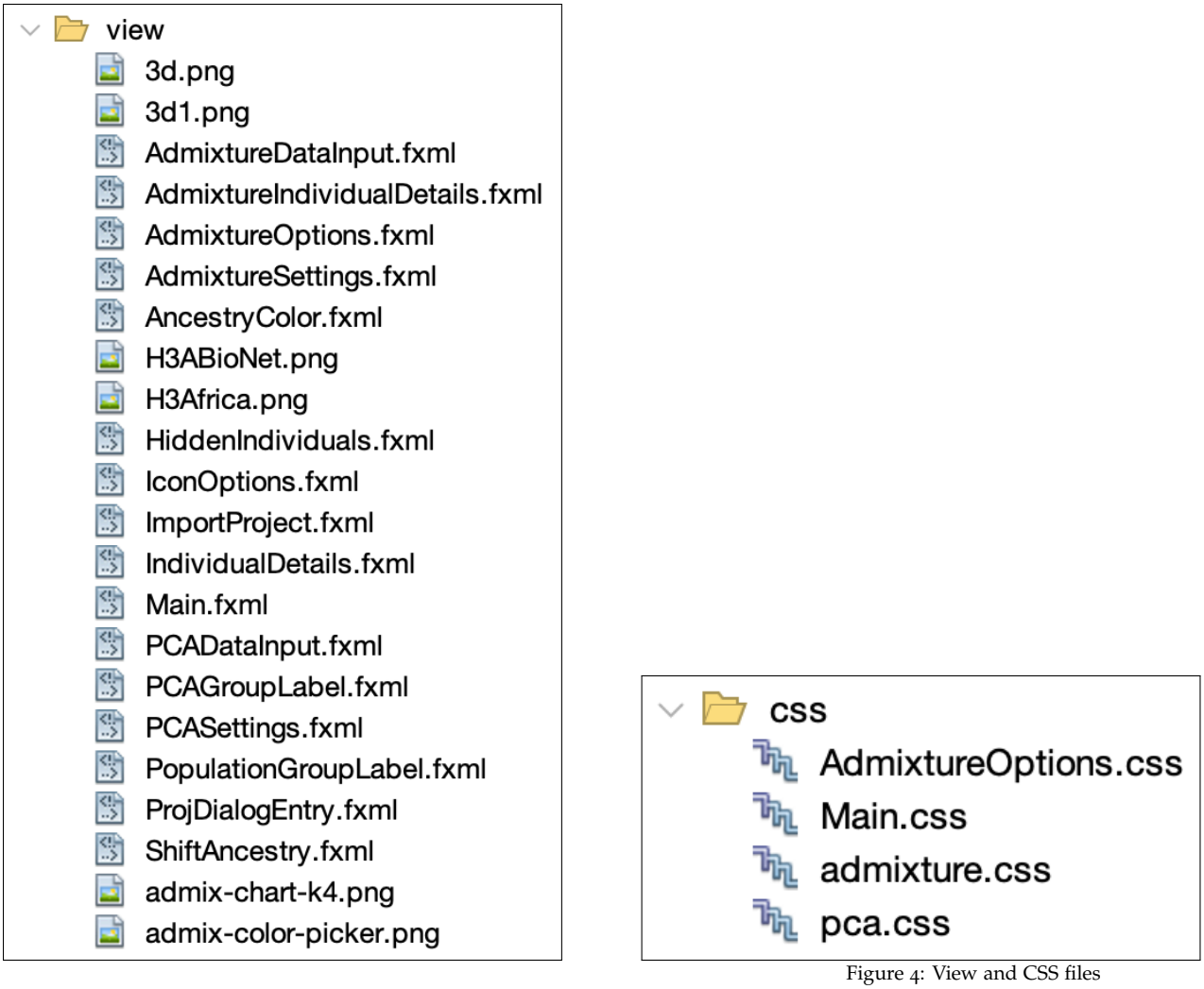

Figure 4: View and CSS files

notation) should also be part of the View.

Only the View and Controller should refer to user interface coding; the Controller should connect user interface elements and the Model, in a way that separates the Model from the interface (View). Saving state should be purely dependent on content contained in the Model, which should reflect the state of the screen presentation as well as underlying data (some of which may not be presented but could be, or is part of the underlying algorithms).

Finally, when running the program from the command line, you get a message like this:

```
WARNING: Unsupported JavaFX configuration: classes were loaded from 'unnamed module @3359c0cf'
```

This does not result in any runtime problem; I found a detailed description[25] of the problem but ran out of time to work through solutions (essentially, adjusting `pom.xml` to find JavaFX libraries correctly).

[25] https://stackoverflow.com/questions/67854139/javafx-warning-unsupported-javafx-configuration-classes-were-loaded-from-unna

# *Problems with the implementation*

The code in inherited was not fully functional. For a start, the development tool chain was not documented so I had to work this out and set the code up to run in NetBeans. This was nontrivial as I had to work out how to create dependences to ensure JavaFX was included. I also had to work out that Scene Builder was the right tool for working on the interface FXML content. Fortunately once I had this set up, building worked; it just took a while to work out all the issues. For example: I had to discover how to configure the `pom.xml` file so that resources were correctly loaded (images, FXML files, etc.).

Another problem was that the built-in user documentation was formatted as HTML files that were designed to open into a web browser. As configured, it only worked out of the IDE; a standalone app did not correctly serve the HTML content to a web browser. It is possible that I could fix this but requiring a web browser is another dependence. For the short term, the HTML help files are accessible from the previous developer's branch on GitHub[26]. These HTML files are a reasonable approximation to the functionality actually implemented.

[26] https://htmlpreview.github.io/?https://github.com/h3abionet/genesis2/blob/henry/src/org/h3abionet/genesis/help/home.html

There are a number of architectural issues with the code. As noted before, it is not a correct implementation of the MVC design pattern. It is instructive to list how many JavaFX imports are in each Model file:

- `AdmixtureGraph.java`:19
- `Annotation.java`:1
- `Graph.java`:0
- `IconsAndColors.java`:0
- `PCAGraph.java`:18
- `PCAGraphLayout.java`:1
- `Project.java`:4

- `Subject.java`:0

For example, in `PCAGraph.java`, all but one of the JavaFX imports are interface-related and hence should not be in the Model layer. Only in three files is the count zero.

Another problem is extremely convoluted data structures. For example, to swap the order of admixture charts, I had to identify 6 different data structures in the `AdmixtureOptionsController.java` Controller file and 4 in the `AdmixtureGraph.java` Model file to swap. A more reasonable implementation would be one data structure in each of these two MVC layers where the order of the displayed graphs would be swapped.

Another issue is that where MVC separation is observed, it is clumsy. Annotations are all recorded in a single giant class representing every detail of all possible annotations that is serialized for saving to a file. A circle, for example, is represented by its centre, radius, line thickness, line clour and translation from its starting point. A text annotation is represented by its start position, text it contains, text colour, whether bold or not, font and translation from its starting position. Clearly there is very little overlap between these two types of annotation. It would make more sense to have a top-level abstract annotation class containing the minimal overlap between all annotations and to derive a class for each annotation type.

Fixing annotations so all details worked – moving one, changing its attributes, cancelling a change and having all such attributes correct across save-quit-load turned out to be a bigger challenge than I expected, for reasons I to into below when I explain JavaFX challenges. I therefore did not redesign the Model's annotation class.

# *Lessons from Understanding JavaFX*

JavaFX has some excellent features but like any library designed for cross-platform development, it has weaknesses. Even aside from the fact that it is no longer part of the standard Java deployment, it has issues that do not quite work the same way on all platforms, or that are not as you expect on a particular platform.

Three examples: a dialog such as open file should stay in front; even a slight touch on a mouse or trackpad at the wrong moment can bring the main window to the front. Consequences of this can include quitting the application failing after the main window closes. Another is in the way opening a file in the operating system user interface works. In the version I inherited, when a PDF of the PCA view was created, it was opened. This works on Windows and macOS but on Ubuntu, it fails. The relevant code is in the Controller class in `PCAGraphEventsHandler.java` and this is the code that fails on Ubuntu:

```
Desktop.getDesktop().browse(file.toURI());
```

It should be possible to check if this functionality is supported using

```
Desktop.isDesktopSupported()
```

However this did not work for me. The final example: opening a file allows you to specify file types as suffixes (extensions). The following code

```
FileChooser fileChooser = new FileChooser();
fileChooser.setTitle("Save chart");
FileChooser.ExtensionFilter pngFilter = new FileChooser.ExtensionFilter("png", "*.png");
FileChooser.ExtensionFilter tiffFilter = new FileChooser.ExtensionFilter("tiff", "*.tiff");
FileChooser.ExtensionFilter jpgFilter = new FileChooser.ExtensionFilter("jpeg", "*.jpeg");
FileChooser.ExtensionFilter pdfFilter = new FileChooser.ExtensionFilter("pdf", "*.pdf");
fileChooser.getExtensionFilters().addAll(pngFilter, tiffFilter, jpgFilter, pdfFilter);
File file = fileChooser.showSaveDialog(null);
```

should return a file name that has one of the extensions offered unless the operation is cancelled. However, on Ubuntu, the user can

type any name and the extension is not appended based on the type selected. This is problematic as subsequent code relies on this; if you do not give an extension the code recognises, it silently fails to save an image file. I did not fix this as I could not find a solution; if the file selection dialog does what it is supposed to do, there should never be a situation where the file name is returned without one of the allowed extensions. it would be best to fix this problem, rather than to create an error or warning dialog[27].

[27] A possible fix could be here: https://stackoverflow.com/questions/32757375/javafx-filechooser-filefilter-not-returning-extension.

An issue I ran into is that manipulating attributes of shapes does not work consistently, which would make it difficult to encapsulate all shapes (in the Controller, not the Model) consistently. Two examples: for lines and arrows, recording rotation is done as a separate add-on transform of class `Rotate`[28], whereas translation (moving the entire annotation) is accumulated as an attribute of each object.

[28] https://docs.oracle.com/javase/8/javafx/api/javafx/scene/transform/Rotate.html

An example of rotation coding is:

```
Rotate rotate = new Rotate();
rotate.setPivotX(arrowAnnotation.getEndX());
rotate.setPivotY(arrowAnnotation.getEndY());
rotate.setAngle(arrowAnnotation.getRotation());
arrow.getTransforms().add(rotate);
```

In the Model, I accumulate the angle rotated in total whereas the transforms applied in the Controller accumulate as separate data structures. Doing it this way could have some utility e.g. in implementing undo functionality. I nonetheless managed to implement the functionality in the Model in a way that is independent of this type of coding and while doing that, I worked out how to show the rotation amount in the user interface as the cumulative change. In the original code, each time the edit dialog popped up, it showed current the angle of rotation as zero, not taking into account previous rotations. The same strategy worked for lines and arrows.

Translate functionality is a mystery. For most shapes, translation, despite being a property of the shape not a separate added-on data structure, worked cumulatively (a new translate amount added to the previous coordinate shift). As with rotation, I added the cumulative amount into the Model. However, with class `Circle`, it is different. Each new call of translate is relative to the starting point.

Little inconsistencies like this make it harder to implement shapes in the Controller or View version of the universe as a top-level abstract class with each shape inheriting from it or another shape that has similar attributes. Consequently, there is some repetitious coding in both the `MainController.java` file that invokes shape-specicific code and the individual class files *`Shape`*`Options.java` (where *`Shape`* is one of `Arrow`, `Circle`, `Label` or `Line`).

# *Major Updates and Outstanding issues*

I FIRST SUMMARIZE IMPROVEMENTS, bug fixes and feature extensions then go on to issues still to be resolved. I end with thoughts on the weaknesses of the programming model used and possible better alternatives.

## *Updates and fixes*

When I took over the project, how it was built was unclear; this documentation as well as the `README.md` file on GitHub fill that gap. In principle it should be a lot easier for someone else to take it over. For example, a complete `pom.xml` file that works for resolving dependences and for installing resources like images into the Jar file is now included in the project.

I found that the `Arrow.java` class as lifted from StackOverflow[29] code and documented that (such code is by a CC BY-SA 4.0 Creative Commons license, so it can be used with attribution).

[29] https://stackoverflow.com/questions/41353685/how-to-draw-arrow-javafx-pane

For all annotations, I worked through issues that fixed the following issues:

- if the dialog for fixing them was in use, clicking **Cancel** did not work; it does now
- if an annotation was dragged, it's new position wasn't recorded properly, resulting in problems like **Cancel** not working
- saving annotations generally did not work

Another thing I fixed was resizing the main window: while this could be done before, contents did not scale with it, resulting in annotations shifting from where they were placed. Resizing the main window now resizes the contents. Ideally, the aspect ratio should be maintained but the saved properties are not altered nor is an exported image, so this is an aesthetic rather than functional issue.

Correctly saving changes to admixture charts did not work – I fixed this so that if the order is changed, this is correctly stored when

the project is saved. This turned out to be very tricky because of the absurdly high number of data structures used to represent admixture charts.

PDF representation of annotations in admixture charts had an incorrect aspect ratio (circles displayed as ovals); I corrected this by adjusting the hard-coded numbers but a better fix is required to make this consistent with the shape of the drawing area.

I also added a check that files read to create a PCA chart have at least two columns and that all rows of a given file have the same number of columns.

## *Outstanding issues*

I would like to ensure that dialogs once opened stay on top; a stray click in the wrong place can e.g. put the main window on top. A fix I briefly explored[30] did not work. It is annoying that this is not simple to do as it is pretty obvious functionality.

Another problem that should not be hard to fix[31]) is the aspect ratio of a PDF image generated from admixture charts. In the code I inherited, the vertical and horizontal dimensions were hard-coded; I altered these to get closer to making e.g. a circle appear round.

The biggest single issue is that the separation between Model and View is weak – Model classes contain a lot of code that is properly part of the View or Controller.

In the process of sorting this out, it will be necessary to alter the file format. When Java serialising is used, a following magic number embedded in the serialised class, as in this example, needs to alter so that incompatible data formats are not read:

```
private static final long serialVersionUID = 2L;
```

In all serialised classes, this constant is currently the same.

In order to do a major change in data formats, conversion would be required.

## *Recommendations*

I recommend working on producing a more maintainable version. The last release I produced is usable and worth releasing as a public beta. However, there is much about the architecture that is still problematic. It was necessary and useful to get it working in its current form to understand better how the whole thing works, but a better architecture would be more maintainable. I spent about 15 hours fixing glitches in reordering admixture charts and much of that was because of convoluted data structures.

[30] https://stackoverflow.com/questions/56084382/how-can-i-make-javafx-filechooser-alwaysontop-of-window

[31] In Controller source file `AdmixtureGraphEventsHandler.java`

A big part of arriving at a more maintainable implementation is cleaning up the MVC design pattern. The Model should contain no JavaFX classes. Any code requiring user interface classes should be moved to the Controller or View. There is no reason that the View should only contain FXML and image files. However, some dialogs that are hard coded would be better implemented in FXML if practicable.

Another aspect is looking at hoe to abstract the different types of annotation so as to create a class hierarchy.

Tidying up the code for admixture graphs would be worth spending time on; ideally there should at most be one data structure used in the Controller and View and another in the Model.

I also recommend reconsidering the use of serializing to create data files. This is a fragile representation. If there are errors in creating the data or if there are objects of classes that should not be serialised, it is tricky to recover the data. If, instead, the data was saved in a human-readable format, the issue of having to maintain magic numbers and a possibly tricky task to convert between versions would be easier to manage. It would also be possible to read the data in a plain text editor or to process it with scripts or other non-Java tools.

### *Weakneses in the programming model and other options*

In conclusion, there are lessons to learn not only from what could have been done better in this example but more generally with the MVC pattern as implemented using JavaFX.

I have already noted excessive use of JavaFX classes in Model classes; for true separation of the model from implementation, there should be none. However, going on from there: how should the Model be implemented to achieve true independence from the user interface layer? That goes beyond avoiding using interface-specific classes in the model but also the issue of a file format that is reusable with other tools. That issue takes me to whether serializing Java classes is a good idea, as hinted above.

why is serializing problematic?

Serializing has the advantage that it captures the state of Java classes in a file. It has the disadvantage that it captures the state of Java classes in a file. What do I mean by this? If you get it right, it's great. What you read is is exactly what was represented in the program previously. This results in a few problems and a few advantages.

First, the advantages: if you code everything right, it's simple. The entire code for saving the project file – excluding a little setup – in

`MainController.java` – has very little to it besides the user interface to ask the user for a file name:

```
File projFile = fileChooser.showSaveDialog(stage);
if (projFile != null) {
    int pos = projFile.getName().lastIndexOf(".g2f");
    if (pos > 0) { // file name ends with .g2f -- cannot just be .g2f
        project.setProjectName(projFile.getName().substring(0, pos));
        FileOutputStream fileOut = new FileOutputStream(projFile);
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(project);
        out.close();
        out.close();
        fileOut.close();
    }
}
```

The most critical lines are:

```
        FileOutputStream fileOut = new FileOutputStream(projFile);
        ObjectOutputStream out = new ObjectOutputStream(fileOut);
        out.writeObject(project);
```

and this 'just works' if the object `project` is serializable (which requires that any fields its class contains are also serializable; reading in is the reverse and also 'just works'. Here is the main code for doing so from `importProjectController.java`:

```
    FileInputStream fileIn = new FileInputStream(projFile);
    ObjectInputStream in = new ObjectInputStream(fileIn);
    proj = (Project) in.readObject();
    mainController.setProject(proj);
    in.close();
    fileIn.close();
```

So far, so good. But what happens if you change a class? There are some changes that do not (necessarily) break serialization. For example, if you add a field then read a file that did not include it, provided the default initialization for it works, you are fine. However if you change the class hierarchy, you will run into problems[32]. So in the situation with the code I inherited where the class hierarchy needs work, you would need to do versioning and data conversion.

[32] https://docs.oracle.com/javase/6/docs/platform/serialization/spec/version.html

Versioning in any class that `implements Serializable` starts from a magic number such as:

```
    private static final long serialVersionUID = 2L;
```

If you create an incompatible version, you need to change this number. Each class can havea different number; if you try to read a class

with a different number than the target class, an `InvalidClassException` exception will be thrown[33]. This looks all good and under control but the problem comes with making major changes that would break serialization. In that situation, you need a version of the original class that can read in the old format and convert to the new, meaning that the new and old classes should be part of the same program,

This could be achieved several ways:

- include both classes in the program – possibly phasing the older one out eventually
- use a conversion program with both classes that reads the old format and writes the new one to a file
- use a conversion program that reads the classes and writes a file in a universal format (e.g., plain text)

The last idea brings me to another drawback of serializing: the output format is inscrutable. Unless you can read it into objects of the same class as the original and convert the data to a readable format, processing it in any other way is difficuly. You would need to know how Java classes are serialized.

If you do have a way to write the data to a plain text format (possibly structured, e.g. XML), why not do that – and make that the main file format?

There are two downsides to a plain text file format: accurate representation of floating-point types and representing complex data structures.

Floating-point numbers converted to decimal lose precision. However, it would be possible to convert them to a hexadecimal representation of their binary form; for human readability, repeating them in character format representing decimal would be useful. This short program[34] illustrates how such formats can be read from a string (that could previously be read from a file then processed further):

[33] https://docs.oracle.com/en/java/javase/17/docs//api/java.base/java/io/Serializable.html

[34] Adapted from https://www.geeksforgeeks.org/scanner-nextfloat-method-in-java-with-examples/.

```
import java.util.*;

public class test {
    public static void main(String[] argv)
        throws Exception
    {

        String s = "Gfg 9 + 6 = 12.0 0x1.921fafc8b007ap+1";

        // create a new scanner
        // with the specified String Object
```

```
        Scanner scanner = new Scanner(s);

        while (scanner.hasNext()) {

            // if the next is a Float,
            // print found and the Float
            if (scanner.hasNextDouble()) {
                System.out.println("Found Double value :"
                                   + scanner.nextDouble());
            }

            // if no float is found,
            // print "Not Found:" and the token
            else {
                System.out.println("Not found Double() value :"
                                   + scanner.next());
            }
        }
        scanner.close();
    }
}
```

Output from this code

```
% javac test.java
% java test
Not found Double() value :Gfg
Found Double value :9.0
Not found Double() value :+
Found Double value :6.0
Not found Double() value :=
Not found Double() value :12.0
Found Double value :3.141592
```

illustrates that any number that can reasonably be interpreted as a `double` in text format can be read as such, *including* a hexadecimal representation of the binary form (in this case, `0x1.921fafc8b007ap+1` – a format designed to make it possible to recreate an IEEE double-precision of floaring point number with the exact same bit pattern as the original, even with differences in byte order on different machine architectures)[35]easy:

```
double pi = 3.141592;
System.out.println (Double.toHexString(pi));
```

and output

[35] An IEEE double-precision floating-point number uses 52 bits for precision, hence 13 digits in total after the "`0x`"; a minus sign is represented as "`-`", rather than as the sign bit, and the exponent uses 11 bits in biased format, but is diplayed using a "+" (optionally) or "`-`".

```
0x1.921fafc8b007ap1
```

is exactly the bit pattern in the full Java program above. Though I illustrate its use in printing out a line, `Double.toHexString` can be used wherever you need to convert a `double` value to a `String` representing its bits as a hexadecimal value.

So with this information, it should not be hard to output text in human-readable form but also with hexadecimal representation of `float` or `double` numbers where precision is required. This format should be readable in other languages, e.g., in Python, you can use

```
{\tt float.fromhex("0x1.921fafc8b007ap1")}
```

to convert a string from this format[36] and in C99, a similar format specifier is defined[37]. A simple C example illustrates that it works:

[36] https://docs.python.org/3/library/stdtypes.html

[37] https://stackoverflow.com/questions/4826842/the-format-specifier-a-for-printf-in-c

```
#include <stdio.h>

int main () {
  double value;
  printf("%a\n",3.141592);
  scanf("%lf", &value);
  printf("%a=%f\n",value, value);
}
```

The same code also compiles using a C++ compiler, with the same output:

```
% ./test
0x1.921fafc8b007ap+1
3.141592
0x1.921fafc8b007ap+1=3.141592
```

For complex data structures, converting them to lists or arrays would make the Model format easier to write to files and read back. Ideally this should be done as simply as possible. XML allows for complex data structures to be represented and, as we know from LISP, lists can represent complex data structures included nested or recursive structures.

However, a simple data format with headwords indicating what data follows and either a count of data elements or a delimiter to indicate the end of a list or similar structure would be simple to implement.

Aside from human-readable files, there is another big advantage to storing data in plain-text (or structured plain text) format: the true independence of the Model from the View is asserted. It becomes relatively easy to process the saved model using other tools (scripts etc.).

In conclusion it would be worth investigating alternatives to serializing objects to store the data. Not only would the data be less inscrutable but it would be possible to import it to other tools and it would be simpler to transition to new data and hence file formats. To do so would be more in the spirit of the MVC design pattern in the sense that the Model would be completely independent from use of the data to present a user interface.

# *References*

Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the Smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

Paul Tyma. Why are we using Java again? *Comm. of the ACM*, 41(6): 38–42, 1998. DOI: http://doi.org/10.1145/276609.276617.

# *Index*