**Recreating Art: An Evolutionary Computation Perspective**

**By Gonzalo Garcia-Montero Barrera**

****

**I certify that all material in this dissertation which is not my own work has been identified**

**Contents**

**Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1**

**Specification**

**Etc…**

**Final report structure:**

* **Introduction** to topic (+ motivation of project, high level picture of the project)
* Literature Review and Project Specification revised summary
* **Design**
  + GUI
  + Painting mechanisms
  + Evolutionary computation
* Development Process
* Testing(?)
* **Final Product** 
  + Description
  + Evaluation
* Conclusion & Critical assessment

**Recreating Art: An Evolutionary Computation Perspective!**

**Introduction**: (*intro to topic + motivation + high level picture)*

Evolutionary algorithms are algorithms based on the phenomenon of natural selection and evolution. These algorithms have been proved useful in problems separate to evolution, that usually include a large search field and whose result is not easy to pinpoint (e.g. fastest path from one point to a destination point). The aim of this project is to point EAs in the direction of art, and explore the ways in which EAs can be useful in producing results.

The motivation for this project comes from my interest in evolution, in natural selection and how critical it is to biology and nature. In my opinion, its way of unfolding naturally in biology seems to suggest it is the fundamental algorithm of nature to yield results. This is why I think it is interesting to mix this component with art, which is another field that I am interested in.

Throughout history painters and artists have been recognised for their creativity, technique and portrayal of emotion. With this project I aim to complete an automated version of the technique aspect, by using automated EAs to generate an image. The creativity side will be up to the user of the system, who can choose an original image for the program to copy and render (this will be fairly simple). The product will therefore be a built system that can successfully paint a copy of an image, by using an alternative painting technique that uses natural selection to evolve paint on a blank canvas, and through a defined fitness function results in a new version of the original image.

**Design**

The first step to designing the system is picturing how I wanted the program to behave. My aim is to have a Graphic User Interface through which the user can interact with the program. The requirements for the GUI are detailed here:

* Allows the user to select an original image from the hard drive, that the program can then use for the fitness function.
* Has a few buttons to allow the user to select the parameters for the evolution (these include things like termination condition, size of the population or elitism)
* Displays the original user-selected image
* Displays a canvas of the best fit candidate in the current generation (updates in real time)

To compare the current design with what I had originally planned, I will attach a version of the literature review & specification that included the primitive version of the specification:

***Old Specification***

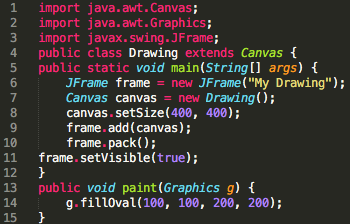
*This section will introduce the requirements for this software, including the functionality and the non-functional attributes of what will be produced. Towards the end we will provide criteria for evaluating the success of the final product.*

*Recap*

*To understand the requirements for our project, we must refresh our memories with our core purposes. The aim of this project is to create a program that a user can easily run to observe how a blank canvas transforms into a piece of art. We want to achieve this with efficacy and a user-friendly Graphic User Interface (GUI). We want to do this in a unique way to have a fun application. That is our goal.*

***3.1 Software***

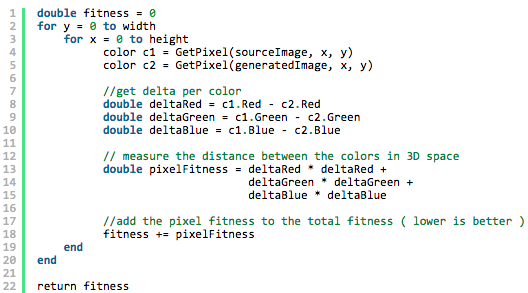
*The programming language we will use for this project will be java, as it has useful libraries for virtual drawing and can easily allow for a GUI to be created. For drawing we can use a simple package for drawing 2D graphics in java called java.awt. AWT stands for “Abstract Window Toolkit”, which includes java.awt.Canvas and java.awt.Graphics, with which we will be able to manage a canvas on a frame in which to draw. On the example below, we are using these classes to draw an oval (instead of a polygon), this example was written by Allen Downey and Chris Mayfield (2016).*

**

***Figure 3.1*** *How to draw a circle in java using java.awt (Allen Downey, 2016)*

*We can also draw different sized polygons and lines of different girth, and random curves by implementing QuadCurve2d for example and CubicCurve2d (which allow for more complex curve structure with two control points be made). When I start programming the software, I will explore these possibilities further to decide on the best artistic form. (Oracle, java tutorials, 2017)*

*Roger Johansson’s work becomes very relevant in this section. He has provided useful code on his application, which can give us an idea of how to structure the program. He has used a simple drawing library to draw the polygons, and provides a pseudo example for the fitness function:*

**

*This is a very straightforward approach using RGB pixel-by-pixel comparison against the original image to provide a numerical fitness value. This is a tool we can implement in our program, as it directly feeds off the literal RGB value of the drawing and compares it in a very systematic way, providing a very precise method of measuring the fitness of the drawing.*

*This concept of using evolutionary algorithms is obviously not new, computer scientists like Roger Johansson have used it, and Juan Romero mentions many evolutionary algorithmic artists in his book. Thus the challenge here is to create something original, different from past material and literature. For this reason I will aim to create a simple yet entertaining program, that unlike the others doesn’t aim to create something particularly revolutionary, but instead creates an efficient piece of software that users can enjoy running and understanding.*

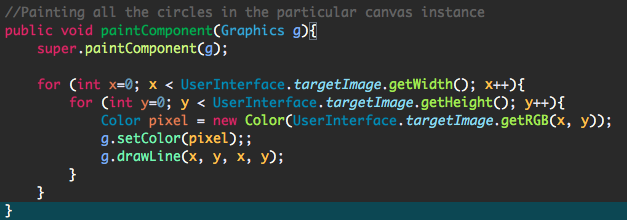
After reviewing the initial version of the specification, it is evident that some aspects of the program have changed through time. At one point when I was considering how to perform the evolution, I was considering doing polygons like Roger Johansson, but decided to not do it so my program would be different to his, and also because it could get difficult having more vertices and painting a polygon rather than painting a circle. I also though of evolving individual pixels, instead of each candidate being a collection of circles, each candidate being a collection of pixels with different colour values. This idea was also ultimately abandoned because the array of pixels would have to be huge (60,000 pixels long, in a reasonably small image) and also because the pixels were too small and simple to evolve. Circles just seemed like the perfect match.

Lets go into detail of how the program is structured:

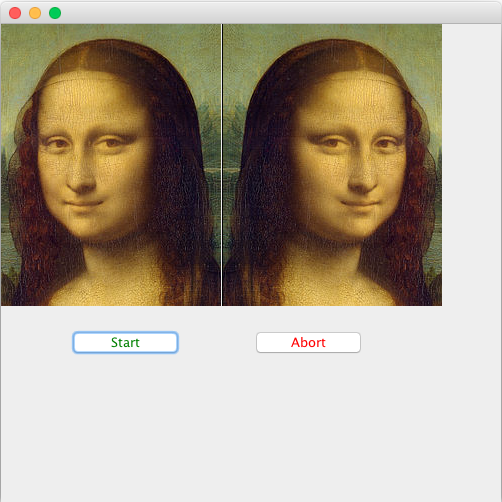
There will be a few classes. These are as follows:

1. User Interface class
   1. This has a few static variables, including the target image as a BufferedImage object and the canvas as a Canvas class object.
   2. A method that reads the image from a particular directory and sets it to the static variable “*targetImage*”
   3. Main, where a User Interface object is created that runs the initialize method (the User Interface constructor runs an initialize method). In initialize we have all the components of the GUI (buttons, JFrame, showing the canvas, the target Image etc.)
   4. Two of the components are a “start” JButton and a “abort” JButton. These have an event action listener that listens for a user click on them. Start() creates a Evolution object and executes it. Whereas abort changes the abort object variable to state aborted (this is a variable that the programme checks as an end condition for the evolution to stop, others can be number of generations, or a specific target fitness for example).
2. Within the interface class, we have a private class Evolution that Extends SwingBackground task (originally it extended swing worker but there were concurrency problems with swing worker). This extends a class from the watchmaker so methods must be overridden.
   1. Methods include perform task, which sets up the different objects necessary for the evolution to take place, including a fitness evaluator, selection protocol etc. An evolution observer is added to output useful information each generation, in this case we update a JLabel with the number of the generation. Finally this method returns a Evolution object with paramaters for number of candidates each generation, the elitism number is defined in the WatchMaker API: *The number of candidates preserved via elitism. In elitism, a sub-set of the population with the best fitness scores are preserved unchanged in the subsequent generation. Candidate solutions that are preserved unchanged through elitism remain eligible for selection for breeding the remainder of the next generation. This value must be non-negative and less than the population size.* I chose 1 for this parameter.
   2. A method PostProcessing() that is called when the evolution is finished. Here it will simply set the abort flag back to non-abort mode.
3. Canvas Class that extends JPanel.
   1. This one has a attribute of a arrayList of painting circles, this array will hold all the circles that should be painted for that particular candidate.
   2. A paint component method (this is better to call rather than paint directly as paint is better for top level containers like Japplet or Jframe) that paints what is indicated inside the method. It iterates through the arrayList of circles and paints each one, by setting the colour to the colour of the circle and painting an oval indicated by the variables of the circle (its X/Y coordinates and width & height). It is important to say that the paint component method is never actually called anywhere, the API calls it automatically (for example when the program is first launched or when the frame is resized). However, to call it purposefully we can use the repaint() method. Note too that inside the paintcomponent method there is an if statement checking if the circle array is empty, this is because the method is always called when the program is first launched, and naturally the arraylist will be empty.
   3. There is a setCircles() method that clears the arraylist of its circles and adds a new list of circles. This could have been broken down into two methods, one to clear the array and one to add the collection of circles. However, it would not make sense with this implementation as the two methods would always be called one after the other.
4. Circle class
   1. This class is for the circles, and instead of actually being a PNG image of the circle, it is an abstract representation of what the circle would be. Its variables hold all the information we would need to know about the specific circle, it’s X coordinate, Y coordinate, width of the circle, height and colour. This is the information needed for the canvas class to draw an oval.
   2. The methods are self explanatory too, they just allow to return a specific attribute or to set the attribute. The attributes are private so can only be accessed through the methods, which allows the code to be safer.
5. Add Circle Mutation, which extends evolutionary operator (every mutation must do this)
   1. This class overrides the apply method, that takes in the current generation’s population as the argument and returns a mutated version of the population. In line 18, the “if” statement contains a way of allowing the code inside the if statement to be executed only with a 2% chance. Rng.nextDouble() returns a random Double value between 0 and 1, and if the value is less than 0.02 (which should be 2% of the time, within the pseudo correctness of the random number generator). Inside the if statement we just create a random circle and add it to the list of the candidate.
6. Circle colour mutation is slightly more complicated than the last mutation class. Again it takes in the population as the argument as and returns the mutated population (as does every mutation). Here instead we have chosen a 1% chance of applying the mutation, but this time the whole population gets chosen for the mutation.
   1. The mutation is carried out by a mutate colour method that takes in a circle as an argument and returns a mutated circle. It mutates every component of the colour (Red, Green, Blue and Alpha). The amount by which it mutates is of up to 20 units (with the 0 – 255 that there are). It does this by using a Mersenne Twister random number generator (I used this one because apparently it is a reasonably fast and accurate one, and in this program fast computation is valued) to create an integer between 0 and 20 (exclusive), by putting 20 as the argument for the method. This color then has a 50 chance of being added or subtracted, otherwise circles would get bigger over time (it also definitely gets subtracted if the value is already the max, 255). The method returns the new colour.
7. Circle Size mutation has a 3% chance of mutating the population.
   1. The apply method first duplicates the circle in the given candidate list, so as to not affect the parent candidate. Then it is mutated and returned.
   2. Mutate size works similarly to the mutate colour, but this time it will subtract the mutation amount only if the circle’s coordinate is out of the canvas by checking the height and width of the canvas.
8. Move Circle Mutation is similar to the other mutation classes we have seen.
   1. As always, it has the apply method, that mutates every candidate solution within the population. Mutated candidates is an array list of all the lists of circles, an array that can be considered the whole population. Later, for each of the candidates, a new mutated candidate is created (the newCircles counterpart), and is populated with newer mutated versions of the circles in the original candidate.
   2. The other method it has is move circle, which is quite self-explanatory. Again it uses a random number generator to create a mutation amount from 0 to 10, and applies it to the circle. Some error handling takes place also to ensure that circles are being drawn entirely within the confinements of the canvas, and not outside of it.
9. Remove Circle Mutation is the final evolutionary operator which simply iterates through the population, with each one having a 2% chance of being mutated. The mutation is applied to a duplicate array list of the candidate, and removes a random circle from the candidate, by using a random number generator to return a number between 0 and the size of the candidate, and remove the circle at that number in the array list.
10. The Circle Factory class extends the watchmaker’s abstract candidate factory class to create random circles, and to also create an initial population of candidates. The method generate random candidate generates a pseudo random candidate of two circles (which is the minimum number of circles accepted in one given candidate), and each circles of parameters chosen by the generate random circle method
    1. The create random circle method is fed a RNG and returns a random circle (within certain limits)
    2. The values for RGB can be any value between 0-255, whereas for alpha the value can only be between 10-127. This is because alpha is the channel for the opaqueness of the colour. If we set it at 255 the circle will be too opaque and will not allow construction and adding of circles on top of it. The opposite is also avoidable, we don’t want the circle to be completely invisible, so after generating a few circles and seeing how opaque they were, I decided to settle for 10 as the minimum opaqueness level.
    3. The coordinate at which the circle is painted is decided from a pseudo-randomly generated number between 0 and the image’s length or height.
    4. The width and height are randomly generated again but this time for a value between 20 and 50. We don’t want the circle to be too massive or too small, and after testing the program a few times myself I decided this was an appropriate level threshold level.
11. The class to judge the fitness of a candidate is the fitness function class. This is one of the more complex classes which took me a long time to compose.
    1. The get fitness method basically paints on the canvas the circles from the given candidate, creates a buffered image out of the canvas and compares the RGB values of the target image against the canvas image. The reason we need to create a buffered image out of the canvas instead of using the canvas directly, is that canvas extends JPanel, and JPanel has no methods for returning its RGB pixel values. This was another problem that I found difficult to overcome as a JPanel object can’t just be cast into a buffered image. Over time I eventually found online a way to get around it, which was by creating the function createImage that used graphics to create a buffered image with the byte information from the canvas.
    2. The generate fitness method takes two images as arguments, (target image and candidate image) and returns the total ‘error’ or variance in RGB values between the two (in a 220x280 image this can be about 150,000,000). The method accumulates the difference in value between the different pixels in the variable fitness, and adds to it the difference in red value, then green and then blue (alpha is not measured as the target image will not have a value for this component). The component difference is measured with the get delta component method.
    3. The get delta component method does a similar principle to the generate fitness method, but actually implementing the work. It takes a string as an argument to determine whether it should measure red, green or blue difference (each time the method is called it will only measure one). The method iterates through all the pixels in the target image, and as the canvas will be of the exact same size, the corresponding pixel will be the same one on the canvas (so for pixel 0,0 it will grab the one furthest up to the left for the target image, but it can also retrieve the pixel in the same position for the canvas). So at each coordinate it makes a colour of the target image \*and\* of the canvas, and depending what string was fed to the method as an argument, the switch will add to delta either the red difference, the green or the blue. This goes on for all the pixels in the image.

It is worth noting that all this design has been meticulously planned after having made an initial program that did not make use of the watchmaker framework. Within this first program that I made, my intention was to use swing worker (similar to how I use swing background task now) and carry out the computation there, instead of in the GUI thread, as it can become unresponsive. This was one of the problems I was having originally, as when I was starting to develop the program, I made the canvas and tried to do simple tasks on it to get the feeling of it and understand it a bit better. One of the first tasks I set myself to do was to simply copy the target image completely in the canvas, which although it might seem easy, it was not. There is no setRGB() method with JPanel like there is with BufferedImage, so the equivalent I tried to come up with was draw line. This method draws a line on the canvas from point A to point B of the colour that you choose. I thought that I could potentially make point A = B so that it would just draw a one pixel long line, which functionally would be the same as a setRGB() method. I tried doing the following:



Which at first glance, it compiles and runs, right? Well, its more complicated than that. From that point onwards, a quest of finding out what the GUI threads do and the UI delegate started for me. I read many oracle java docs on the 2D graphics supported by these classes and how the threads of the GUI worked, but I couldn’t really grasp exactly why it couldn’t work. In the end, the GUI thread (the thread that executes the GUI and the JFrame) has a few UI delegate threads that are spread to carry out its tasks. However, these aren’t prepared for vast amounts of computation like what is needed to iterate through all the 60,000 pixels in an image, especially not in the paint component method. However, I realised that if I set a button with its event action listener to paint one single pixel every time its clicked, that did work. Obviously you would have to click it 60,000 times if you wanted to display the whole image, but at least I learnt a bit more about how the GUI delegates work. After having practiced for months, I now know that the best way to achieve this is call the method from a swing background task object to do the hard work, and thanks to this I could achieve an image like this:



by displaying the X coordinates in its inverse, to achieve the reflection of the mona lisa.

Also when I was learning how to read image files in java, I learnt how to create and write buffered images to the disk, and I achieved a program that could take a normal image as an argument and write the same image but in negative (I had to research that a negative image is made through getting the RGB values and subtracting them from 255).

After actually making a whole program to simulate the evolutionary process, I designed so that instead of one class for each mutation, I would have one method for each mutation, which would basically perform the computation that the classes do now. All this computation would be done inside a swing worker object, and there would only be two candidates in the population, one would be the previous set of circles, and the new one would be the current set. However, I had huge problems with the generating fitness function, which is actually the exact same one that I have now. The problem rose from the fitness function not actually being able to be output directly from the swing worker object. That will be discussed later on in the paper.

**DANIEL DYER**

After having completed the program using the WatchMaker framework, the evolutions worked correctly, however the problem relied with the selection process of the fitter candidate. The way I wanted to do it was to have a population of TWO: the parent, and the child. The child will have been a mutated version of the parent and therefore slightly different, and after having the two together the selection mechanism should evaluate which of the two are fitter (resemble more closely the target image) using the generateFitness method, for then choosing the fitter one as the parent image for the next generation. I got this idea from Roger Johanson’s work, as in his “frequently asked questions” section of his EvoLisa program he said to have a population of just two, and claimed it was still genetic programming: “*I will claim that this is a GP due to the fact that the application clones and mutates an executable Abstract Syntax Tree (AST). Even if the population is small, there is still competition between the parent and the child, the best fit of the two will survive*”. After pondering on a few different implementations of a selection process, I thought I would try that one too because computational power would be saved each generation if fewer canvases were needed to be rendered.

Another implementation that I had in mind that my supervisor suggested, was instead of considering the painted canvas itself as the candidate, considering each circle/polygon as a candidate. That way their fitness could be measured by simply comparing the RGB pixel values of the images at the corresponding coordinates of each image. This method was dismissed due to two problems:

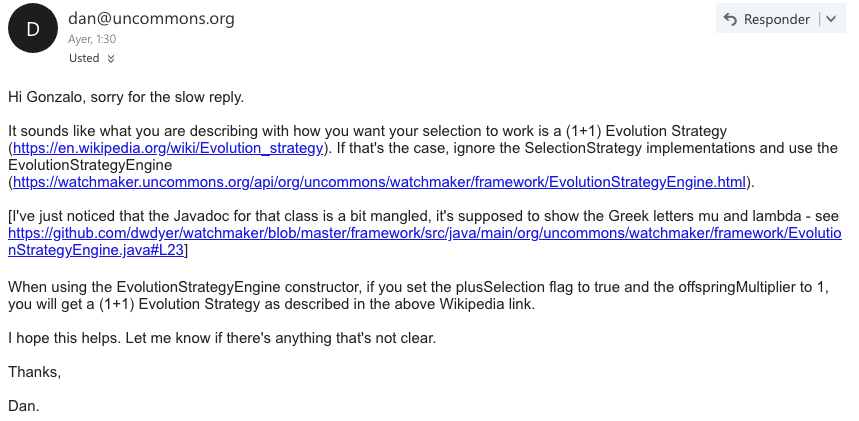
1. The nature of the program allows for translucent/semi-opaque circles so that they can be stacked on top of each other, to give a more varied spectrum of colours and approach the target image more closely. If this method of measuring the fitness of each individual circle was carried out, it would be very difficult to extract the fitness of the individual candidate. as the longer the program runs the more candidates there will be, and the likelier it will be for two or more circles to be stacked within the same space, thus giving an RGB value of the combination of circles within that space, instead of just the individual (which is what we want to measure).
2. The second problem with this method would be that the measurement of the circles would be inaccurate at the corners. The method to compare the RGB values for this implementation would presumably be a version of the generateFitness() method I have already written, by taking in arguments of the coordinate of the circle and its width and height. This method would therefore iterate through a rectangle of this size in the canvas where the circle is, but it would be very difficult to limit the method to just the circle. For instance, in the example below this rectangle is what the method would evaluate, taking the RGB value of the black corners too, instead of just evaluating the circle inside it. The fitness value would become unprecise unless a different, more complicated method is used.

https://lh5.googleusercontent.com/02D-7OT3s7OlrU33aJxdwTKhvydbJtYkg71Yy3m4OxgCB3sQkpXD3EvPSaGNxTYuePoMDbiVSQjiTDtm4m60-iar2n0-DYN5-2m8cl7bGoVTdei51Prr6h9ZWkM15oD2K6j--ifE

After settling for this parent and child population system, the problem I was having was that I could not actually keep the parent image for the selection process. Only the child was retained and evaluated, which was useless as there was nothing else to compare it to. Thus mindless evolution was taking place with no careful selection process to approach the canvas to the original image.

After browsing the watchmaker framework API documentation, I was not clear on what selection strategy to choose to implement this parent/child population method. Online blogs like stack overflow were of little use too as the little popularity of the framework means that not many people have used it and there is not much third party support online.

Thinking outside the box, I decided to go the extra mile and try contacting the author of the framework directly, he would be the optimal person to solve this problem as he would know the framework inside out. In the watchmaker website there is a link to Daniel Dyer’s website, which has a link to his LinkedIn profile, twitter and various others, but however doesn’t show an email to contact him. I decided to try his twitter (even if this was not the most professional of media to contact on) as I had a twitter account. I tweeted him explaining that I was creating an evolutionary program for my dissertation in Exeter, and ask if I could ask him a few questions on how to use it. Luckily enough for me, he replied the same day, his response was “*Yeah, no problem, I’m out tomorrow so might not respond promptly. You can email* [*dan@uncommons.org*](mailto:dan@uncommons.org) *if that’s easier*”. Needless to say, I was thrilled to have gotten Daniel’s attention and to be able to ask him questions about his framework. I emailed him with information about my project and detailing my problem with the selection process and the unresponsibe evolution. His answer was:



Even though his answer was not completely specific to my program, it gave me a better idea of how to implement this selection process. It is worth noting that in the time that Dan took to reply, I was still trying to make the selection process work, and I figured that when I created the evolution engine I was setting the population size as 1. This was okay for what I wanted, the comparison between the parent and mutated child selection, however at this point this selection had not been chosen, so what was happening was that a normal selection process “Roulette Wheel Selection” was taking place. This selection was meant for more than one candidate, and that is why my program was not evolving accordingly. In addition to this, the elitism parameter was set to 0, when it should have been higher so I changed it. After having changed these two parameters, the program should have evolved fine, but as it always happens with programming, the program will do what you *tell* it to do, not what you *want* it to do. The program evolved similarly without progressing much in looking like the target image.

**Process**

The process of development was talked through with my supervisor. It was broken down into simple tasks every week so that it would be easier to develop. Most of this tasks, however, were knew to me. As for example I had only had experience with GUI programming in Software Engineering in a group project where I didn’t contribute to that section of the assignment, so I had to learn a lot of things for this project.

The process started with me making a simple graphic user interface, and after this I could be able to start adding new functionality every week. I Downloaded some GUI design software (links in source code) that allowed a rendered GUI to be visible before compiling the code, to make it easier to design. The type of project was a swing designer window builder, which allows to modify and design the GUI. I had help from the channel ProgrammingKnowledge (2014) on youtube to know how to use it. After achieving this, the next goals were about drawing different components on the GUI, and implementing several needed components. One of the problems I faced one week, for example, was that painted circles had a blank square around them, even if there was another circle supposed to be painted around it. I realised after reading a few more oracle java docs that when you use the repaint() method with arguments in it, it doesn’t only repaint the part of the canvas that you say in the arguments with the computation you arranged (a circle), but it also paints the background colour with the standard background colour you set the canvas, even if that means painting over some other component on the canvas. This was due to on of the UI delegates that did work “behind the scenes”. This problem was solved by calling the no-argument version of the method repaint, however this means that every time that one circle is painted, the whole canvas is painted too, which potentially leads to decreased efficiency. I got around this problem too, by indicating in the paint component that every circle on the circle array list should be painted. Even though this seems to suggest one different line of computation for each circle, in reality, the UI delegate threads put all these commands together and execute the painting mechanism as one, so actually all the circles are painted at the same time (and no blank squares are rendered). This means that the whole canvas is painted over once and there it isn’t painted entirely for each time a circle is rendered.

At one point, after I had a few hundred lines of code I decided it was time to back up my system somehow, and I decided to upload my project on a github repository. This is good practice for developers and I had done it in the past for another one of my modules. After a few classes were implemented in the program, I decided to put the evolution working and it seemed to work at first, but it had some inconsistencies. It was at this point that I decided to focus more on the mutation methods, that I would have to do sooner or later, and leave the inconsistencies for the future. Needless to say, most of what I learnt throughout this process was through oracle documentation, online video tutorials, and developer blogs.

Obviously, throughout all this process I had to balance other modules’ work and coursework assignments, so process had to be digestible. It was agreed that because throughout the term it was very busy, and in April I would have virtually all the time dedicated to the project, I could leave a big amount for then. However, unfortunately I had a health problem in early April that made progress more difficult. This was an unforeseeable problem that I didn’t expect.

**Watchmaker framework**

I found a framework online called the watchmaker framework for evolutionary computation. It was developed by Daniel Dyer in Kent and at first I was sceptical about using, but decided to download it because it included a program of evo-lisa similar to the one I was developing, and it could be helpful to see how he did it. At first, obviously I found it really difficult to follow, as his program was huge and used a framework that I wasn’t familiar with. It was only through weeks of studying it and browsing the watchmaker API documentation that I slowly began to understand it and conceptualise what he was doing. This was slightly more useful than Roger Johansson’s original evo-lisa because his was written in C#, a language I am not familiar with, and I couldn’t find a way to run it either.

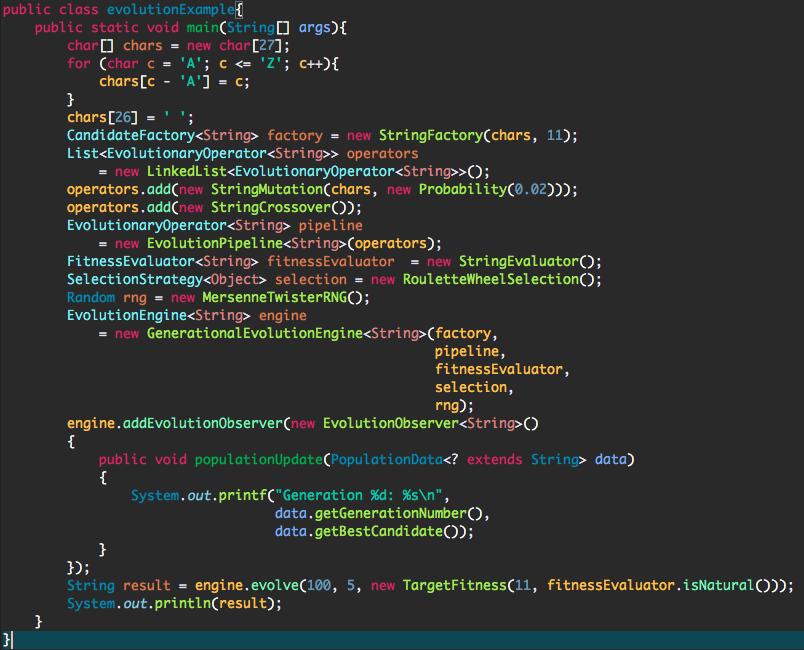
I also found it difficult to understand how the jar file worked and how to download the library. A tutorial online that I found helped me understand it and add it to my personal libraries on eclipse from the cannel Gontuseries “Installation and setup in eclipse” that helped me find out how to do it.

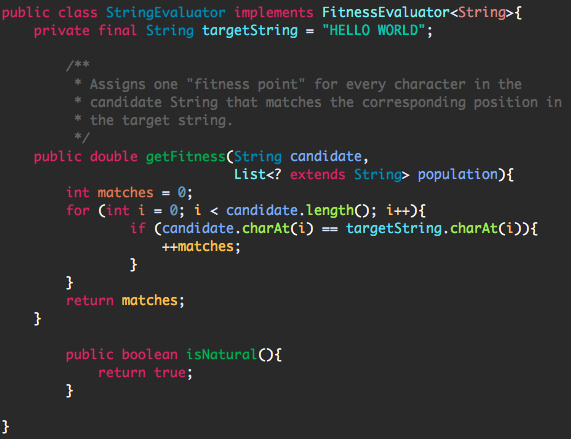
The framework is very useful as it provides interfaces to evolve objects and classes, which makes the whole process neater and easier to understand.

All the documentation about the packages it has and the classes that it includes is available at <https://watchmaker.uncommons.org/api/index.html> which is very useful to understand what I am doing and what functions to use, and to see what the best option for each thing is (for example for creating the candidate factories, I would have to use the abstract candidate factory adapted to my Circle class, instead of using a made one like the string factory).  
 I ultimately decided for the watchmaker framework after the swing worker class simply was not working for my program, it was coming up with too many bugs. Watchmaker seemed like an attractive alternative, which also meant that I could have everything more neatly displayed into different classes, plus a clearly defined evolution engine with its different parameters. Also, after reading through Daniel Dyer’s example program and trying to understand it, I became a bit more familiar with the framework’s classes and functions.

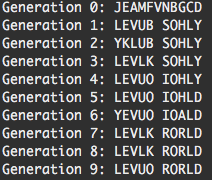
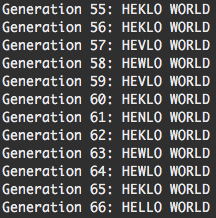
**Testing/Experimenting**

When I first used the watchmaker framework, I created an example program to evolve strings into a specific target string, an example that came with a tutorial in the watchmaker website: <https://watchmaker.uncommons.org/manual/ch02.html> this however was slightly out of date with the exact parameters of the functions and had to read the API thoroughly to get it working, but after a few tries I indeed got a evolutionary “Hello World” program to work!





It outputted a number of random strings that slowly evolved into the hello world string:

This gave me more confidence to jump into the watchmaker framework and start developing my own evolutionary art program.

Created the returnRGB function to retrieve the RGB value from a bufferedImage to later on compare against the canvas and retrieve a fitness function. However, the JPanel class does not have a getRGB method, which was problematic for the design. At this point I feared having to consider a radical new approach to the MyCanvas class, to extend a different class that would allow to getRGB from it (as JPanel cannot be cast into BufferedImage). However, after some online research I found a way of indirectly casting it, by creating a method that could create a bufferedImage object from the MyCanvas, by creating a bufferedImage object whose arguments are the MyCanvas.getHeight() and MyCanvas.getWidth, and BufferedImage.TYPE\_INT\_RGB. The bufferedImage method createGraphics() is used to create a Graphics2D object, which will be used to draw into this image. We then paint the panel into the bufferedImage by calling the paint method on it.   
To test this worked, I simply wrote this code to returnRGB from the newly created bufferedImage from the MyCanvas and print out every single RGB value, to make sure that it worked correctly:

BufferedImage newCanvas = createImage(canvas);

ArrayList<int[]> rgbs = returnRGB(newCanvas);

for (int[] intArray : rgbs){

System.out.println(intArray[0] + " " + intArray[1] + " " + intArray[2]);

}

When trying to read a PNG file from the system, I wrote some code to test the reading RGB values from it and manipulating them, to start having the confidence of using these methods and understanding their function. Here after retrieving an original image from the filesystem, proceeded to create two bufferedImage objects that I would later write out to the disk. One was to make a gray scaled version of the image, and the other created a negative version of it.

BufferedImage grayScaleImage = new BufferedImage(img.getWidth(), img.getHeight(), BufferedImage.TYPE\_INT\_ARGB);

BufferedImage negativeImage  = new BufferedImage(img.getWidth(), img.getHeight(), BufferedImage.TYPE\_INT\_ARGB);

for (int i=0; i < img.getWidth(); i++){

MyCanvas.x = i;

for (int j=0; j < img.getHeight(); j++){

MyCanvas.y = j;

//Get rgb colour in each pixel

Color c = new Color(img.getRGB(i, j));

int r = c.getRed();

int g = c.getGreen();

int b = c.getBlue();

int a = c.getAlpha();

//simple grayscaling

int gr = (r + g + b) / 3;

Color gColor = new Color( gr,gr, gr, a);

grayScaleImage.setRGB(i, j, gColor.getRGB());

Color nColor = new Color( 255-r,255-g, 255-b, a);

negativeImage.setRGB(i, j, nColor.getRGB());

}

}

//this code is to test to create a grayscaled/negative image

try{

ImageIO.write(grayScaleImage, "png", new File("/Users/gonzalo5207/Documents/workspace/EvolutionaryArt/src/grayscale.png"));

ImageIO.write(negativeImage, "png", new File("/Users/gonzalo5207/Documents/workspace/EvolutionaryArt/src/negative.png"));

} catch (IOException e) {

e.printStackTrace();

}

Using watchmaker math framework.

When I was trying to randomly generate population of circles and mutate them randomly, I found myself using the MersenneTwisterRNG (as I read online that this was more suitable for this type of task due to its speed and accuracy of pseudorandomness). At first I didn’t understand why simply creating a new MersenneTwisterRNG object and using that as the randomly generated number wouldn’t work, but after reading the API docs carefully and the oracle docs for the Random class, I found that simply instantiating it didn’t create a randomly generated double, but created the random number generator. To actually get a random double value from the generator I needed to instantiate it each time, also calling either nextInt method or nextDouble to generate a random number. Depending on what I wanted at the time, either method was more useful, but the lucky thing about the nextInt method was that you could put an int as an argument, and this would make the randomly generated number be between 0 and the argument number.

Problem with removeCircleMutation, after trying it for a very long time I came across a null pointer exception. The problem was that I hadn’t planned a case scenario for when the array of circles was completely empty of circles (could theoretically happen if the remove circle mutation probability is set high enough in comparison to the addcircle mutation, and depending on the initial population of circles). Hence, the only thing I had to do to settle this problem was write an if statement at the top of the remove circle mutation method, indicating that if the random circle array was of length 1, the method should ignore any computation and simply return, allowing to give a chance for the next generation to mutate more circles and populate the canvas.

**Inconsistencies in generating fitness:**

I never expected the generate fitness method to work unexpectedly, as it is a synchronised method and its computation is fairly simple (iterates through the pixels of both the original target image and the candidate image and returns their accumulated RGB difference). However, I decided to test this by running a simpler version of the program. This one would have just one circle on the canvas, and each generation would not affect the canvas at all, to keep the fitness identical. After each generation I would call the generate fitness method to return the fitness integer, expecting it to be identical each time. To my surprise, there were inconsistencies with each generation. The inconsistencies range within millions (for a program whose initial fitness is of tens of millions, this is a significant deviation). I didn’t know what approach to take for this problem, as it seemed to be a concurrency problem with the variables (int variables for the RGB values that get manipulated simultaneously) within the method. One approach, is to break down the method into smaller methods, having a method to simply return the delta int value of a component (the difference in red value for example). I got inspiration for this from <https://stackoverflow.com/questions/5861894/how-to-synchronize-or-lock-upon-variables-in-java?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa>

The way this method works, is it takes the two comparing images as argument (the original target image and the candidate image) and the component to be contrasted as a string (red, green or blue). It then iterates through the pixels on the images, and for each one creates a colour. It then uses a switch statement so that depending on the colour argument it retrieves that colour and returns the delta of that colour. This method was tested in the same way the generate fitness method was tested, by putting a non changing canvas through generations and trying to input the fitness each time. It completed the test successfully.

I then tried to modify the generate fitness method so that it would call the get delta component method three times, one for each RGB component and return the sum of the three deltas. Here, I had inconsistencies again, and each time the value displayed was different (though the range was slightly decreased since last time), and inexplicably the problem seemed to only rise with the red component. I figured that out because I modified the code to test exactly where the program was going wrong, and the red component gave differing values. However, it is worth noting that these discrete values reappeared several times. **See photo below**.

<https://stackoverflow.com/questions/8567905/how-to-compare-images-for-similarity-using-java/?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa>

This was really hard to pinpoint exactly why it was happening, and at this point I really struggled to find what the flaw was with the program, I had broken down the computational steps to take into easy-to-digest statements however this problem seemed unfixable. Therefore the steps I took to try and overcome the problem weren’t to fix it, but to adapt the code around it. I modified the generateFitness so that it would call the getDeltaComponent for red multiple times, generating an array of results. From this array, a more appropriate candidate for the red component could be chosen by looking at the array as a whole (by finding the average of all the values instead of just choosing one of the values, which is the result we would have gotten otherwise). This way, I hoped to get more concise values for red. However, I found that after implementing all this it still would give varying values.

I couldn’t find an explanation for this, but what I could observe was the outcome. Red was the first component to be evaluated and would be the only faulty one. This led me to suspect the problem was linked to it being the first component being evaluated. To confirm my hypothesis, I tested the program by calling the three components as usual, but calling red again before the other three, so as to buffer the problems linked with the “first component evaluated problem”. The results are below, they were surprisingly good, the test call would vary with its results but the other three red, green and blue would be consistent.

**Solution:**

After many tries in desperation, the solution became clear unexpectedly after I was trying to email the problem to a professor. To help him help me, a made a simpler program that simply ran the get delta component method with two images, so he could run it, see what is going wrong and give me advice on how to prevent it. To my short disillusionment, the program seemed to output correctly. Every time. This meant that there was another factor outside of the method that was preventing it from giving consistent results in my program. I went back to my full program and decided to run the method outside the java swing worker (the class I was using to run background processes), and it worked perfectly. Therefore, the problem was concluded to stem from the use of the swing worker class, so now that I know that the method works, I simply need to find a way around swing worker so it can be run correctly.

**Results and Evaluation**

(IDEA FOR LATER: INCREASE EFFICIENCY OF IMAGE COMPARISON BY ONLY COMPARING THE SEGMENTS OF THE CANVAS THAT HAVE BEEN ALTERED IN THAT GENERATION, e.g. COMPARING THE CIRCLES THAT HAVE BEEN MUTATED AND ALSO TAKING OFF THE BLACK CORNERS).

**References**

Genetic Programming of Mona Lisa - Roger Johansson, 2008 <https://rogerjohansson.blog/2008/12/09/genetic-programming-mona-lisa-faq/>

Gontuseries – Installation and Setup on eclipse, 2013 - <https://www.youtube.com/watch?v=lw3fPT8whOc>

ProgrammingKnowledge, 2014 – making a GUI on eclipse

<https://www.youtube.com/watch?v=r8Qiz9Bn1Ag>

<http://www.cap-lore.com/code/java/JavaPixels.html> pixel setRGB

<https://docs.oracle.com/javase/tutorial/uiswing/painting/index.html> main lesson

<http://www.oracle.com/technetwork/java/painting-140037.html> painting in AWT and SWING

<https://docs.oracle.com/javase/tutorial/2d/index.html> 2d graphics

<https://www.youtube.com/watch?v=ydQWhluoBXM> 2D graphics tutorial

<https://www.youtube.com/watch?v=X5Q-Mecu_64> swing worker

<https://stackoverflow.com/questions/1349220/convert-jpanel-to-image?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa> BufferedImage from JPanel