Australian National University **College of Engineering and Computer Science**

Research School of Computer Science

COMP6700 Assignment 2

JavaFX and CaptchaSVG

Due 23:59 EST on Sunday, May 25, 2014

No-penalty Extension till 18:00 Monday, June 2

Total mark: 20

Abstract

The second assignment asks you to complete a Java program, called CaptchaFX, which uses the *JavaFX* library for creating graphical user interface (*GUI*) applications and/or (graphically) rich client applications (*RIA*). *JavaFX* is the latest and most modern GUI/RIA API which allows it (or, at least, provides the technological means) to compete with platforms like *iOS*, *Android* and others in the lucrative market of hand-held touch screen devices (smartphones and tablet computers).

The program attempts to create a sequence of visually distorted representation of characters which can be used in CAPTCHA test. Your task is to extend the program by creating a simple user interface with a few control elements ("buttons"), which can be used to control the program behaviour. You should also extend the *Model* part of the program to allow a random selection of characters one at a time from different font sets by the program itself. Finally (and most ambitiously), you should implement one of the simplest CAPTCHA algorithms to make recognition of characters more difficult for an optical character recognition (*OCR*) program, yet keeping it still relatively trivial for humans.

This assignment covers most of the topics we discuss in Blocks 5 (primary topic), but also Block 4 and all Java programming fundamentals. One of the assignment task is to document the newly added code by well-formed doc-comments which can be processed by `javadoc` tool to create detailed and useful documentation. The estimated effort is 12-15 hours of work including studying relevant API and programming techniques.

Background: CAPTCHA

CAPTCHA — **C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part [1] — is a test invented in 2000 by researchers from Carnegie Melon University and IBM. It involves asking a human internet (or any other computerised service which involves granting some kind of access) user to recognise a semantic key — type in a sequence of characters (usually forming a non-word) which are displayed on a screen in a form of distorted image, which may look like this:

.

The idea was that in a distorted form an image of such kind is still easily recognisable by a human, while for a computer (running an image scanner and an optical character recognition software (OCR) which is used for book scanning) that would still be impossible or at least much harder problem to solve.

Alongside with this, text-based CAPTCHA in which the semantic key is a set of standard characters, there are versions based on use of an image as a semantic key (for example, sometimes a downloading of an electronic

[1] So, strictly speaking — CAPTTTTCHA, but no one wants to stutter volunteraly, right?

article from journals published by the American Physical Society are filtered on the ability of the user to select a picture of Albert Einstein from a collection of six to eight images some of which can even display a human person who is not the famous physicist). For the purpose of this assignment, we are only dealing with text-based capture.

The original program attempts to use the *JavaFX* library to generate (not yet distorted) images of letter-patterns which can be used in the captcha tests as the semantic key. The source of letter-patterns is one of the data sets which are stored in separate files (one file for a font) located in the directory `svgfonts`. The files with the extension `.svg` are description of how every symbol (character) from a font set is to be drawn on screen; an image of a character in a particular font is called a *glyph*. The glyph rendering code is written in the so called **S**calable **V**ector **G**raphics (SVG) format, an XML-based resolution independent code which defines how every glyph is assembled by drawing and closing a path element [2]. Like all (good) fonts, the SVG font format also contains information about how glyphs need be placed alongside each other to form an aesthetically pleasing “printed” text. It is not of any importance for you in this assignment, but this is how the code describing rendering of one of the characters looks like (in the font *DroidSans-webfont*):

```
<glyph unicode="w" horiz-adv-x="1528" d="M1008 0l-168 616q-4 18 -10
40.5t-12 47.5t-12.5 51t-12.5 51q-14 58 -29 120h-6q-14 -63 -27 -121q-11
-50 -23.5 -103t-23.5 -90l-172 -612h-211l-281 1098h191l131 -584q10 -45
20 -97.5t19 -104t16 -97t11 -74.5h6q5 26 13.5 69.5t19 92t21.5 96t21 80.5l179
619h196 l172 -619q9 -34 20.5 -80t22.5 -93.5t20 -91.5t13 -73h6q3 26 9.5
69.5t15.5 95.5t19.5 106.5t21.5 101.5l137 584h186l-284 -1098h-215z" />
```

The element `<glyph>` has in its attributes [3] the following two:

- `unicode = "w"` — the character ‘w’ whose glyph description the font contains
- `d = "M1008 0l-168 616q-4 18 ..."` — the path code for the actual glyph rendering

The SVG font reading class uses the so called *SAX* library (part of the standard Java SE API), which can read (as a stream of tokens) the content of an XML file, detect elements with the required attributes, and use an attribute value to extract the corresponding value. You do not need to fully understand the details of how it’s done; for you tasks here, it will suffice to know that thus extracted ‘path code’ is converted into a string object, and this string is used to create a shape object `charPath` (an instance of the class `javafx.scene.shape.SVGPath`) which is added to the scene and made visible.

Currently, the program requires the name of an SVG font file to be passed as a command-line argument. Once the file is read and parsed, the programs “knows” how to render each of the characters from the chosen set. The chosen set for the purpose of this assignment is defined to contain only the **low case ASCII letter characters and decimal digits**.

The user can generate glyphs by typing on the keyboard. If a key representing a character from the chosen set is entered, the corresponding glyph is displayed. The displaying involves scaling the actual glyph to a smaller fixed size, and translating (moving) it to a calculated position to allow successively generated glyphs to form some sort of a line. Only one horizontal line of glyphs can be filled (`CaptchaFX` is not an editor program after all). The generated glyphs can be removed (the scene cleared) by pressing the `ESC`-key. One can also take a snapshot of all glyphs and save it a file by doing double-clicking the mouse.

It does not serve a practical purpose (yet it is not totally gratuitous), but the glyph displaying also involves every glyph appearing at the same location (middle bottom) and ’flying and somersaulting’ to its final location. I added this part (which you need to remove in you final version) to illustrate programming of transition effects. The transition and transform features of *JavaFX* can be of use for creating the glyph distortion.

That’s all the original program can do.

[2] All kinds of two-dimensional graphics can be described in SVG format. Programs like Inkscape, Adobe Illustrator and others can save a created image scene in the SVG format. Animated graphics can be also described.

[3] In XML, elements can contain one or several attributes, each being a pair `name=value`.

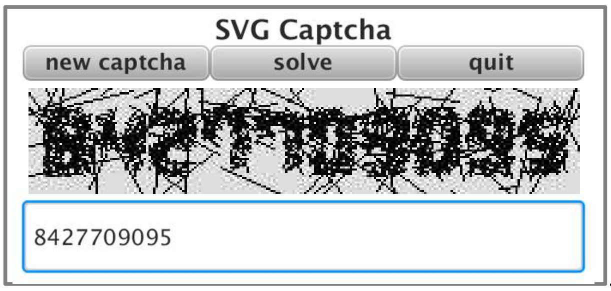
Assignment Tasks

I provide the following scantily documented files

1. `CaptchaFX.java` — the main file which also sets the scene and created all visual effect. From the Model-View-Controller architecture point of view, it is the View **and** Controller component.
2. `svgfontreader.SvgFontReader.java` — the SVG font reading class which contains methods for extracting a code path for a character from the chosen set; the chosen set is defined in this class as well.
3. `svgfontreader.CharNotRepresented.java` — a simple unchecked exception which is thrown on the attempt to get a code path for a character not in the chosen set.
4. an *empty* class `captchariser.Captchariser` — you *may* implement it for implementing the last task of distorting the glyphs, but you may follow a different implementation plan and do the distortion in the main class. I leave this up to you.

Your tasks are to extend both the Model and the View-Controller part to implement the following program features:

1. Extend the Model to allow at every request from the Controller, to return a code path for a *randomly* selected character from a randomly selected font. This means that the model should open, read, parse and store all the characters from all the available font files (their names are to loaded at the start from — you guessed it! the properties file `svgfonts.properties`). The Model should use such added capabilities to construct the semantic key, a sequence of fixed length which contains randomly selected characters from randomly selected fonts (the font value matters, of course, only for rendering the glyph; the value, *i.e.* the meaning, of the key depends on characters only). The number of characters in the semantic key must be fixed (as a `private final` field); its value can 7 or 8 (a simple extension of the program would allow to set the level of captcha during at run-time, but this is trivial, and I am not asking you to do it)
2. Extend (create anew) a simple user interface to operate the creation of captcha images, solving them and quitting the program. This interface can look like this, for example: When the user presses the



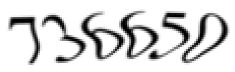
8427709095

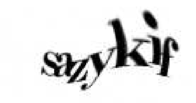
.

`new captcha` button, the program generates the semantic key, the glyphs, performs the distortion (if implemented) and displays them in the sub-window. The user types in the solution and presses `solve` button. The program announces the result (correct or wrong) in a form of a pop-up window (you choose). When the button `quit` is pressed — you may guess what should happen, right?

3. After creating, scaling and moving the glyph to the required position (of in between any of those stages), distort the glyph by using one of the following approaches:
   - introduce pixel noise in glyph image; **important:** in this approach, it may be better to introduce noise at the end, when all the glyphs are rendered in a sub-container — this will allow to get noisy background homogeneity and make glyphs less pronounced individually; the noise can be created by working with the pixel matrix, or (simpler) by adding random graphic elements like lines and other shapes, partly transparent or not
   - perform tilting at random angle and/or scaling up or down of every glyph

- (most challenging) apply the distortion (like a picture on a plastic stretchable surface can be distorted); consider two following options — The left — gradient swirl — requires to perform a

  rotation with sheer which slightly changes with the radius measured from the image centre; the right — multi-waves — can be obtained if to subject the pixel matrix to wavy deformations, both vertical and horizontal. Each of the two requires to work with the `javafx.scene.image.Image` class and read/write the pixel matrix.

4. Lastly, you need to **add** meaningful and properly formatted doc-comments to the originally present code and **include** doc-comments to the most important code which you will write yourself.

Marking Guide

The marks will be allocated in the following way:

5 points — for extending the model to allow random selection of characters for the semantic key, and random selection of fonts for the code path.

5 points — for creating the UI as described above; this involves definition of the call-backs (even if some of them may not be fully implemented, since this is the later task, which you may or may not complete; in the latter case, make the program to report this a call-back stab implementation).

6 points — for displaying the captcha glyphs; this is already implemented by the original program, you just need to adjust the code a little (in particular, remove those flying-and-rotating feature); the captcha should be displayed all at once in a designated sub-window.

1 point — for writing appropriate doc-comments; pay attention to *class headers*, in particular

3 points — for the most complicated task of creating glyph distortion; you will get 3 marks if you choose a simple scheme, like noise of random tilting/scaling or similar (which you may find yourself)

3 points — this is **bonus**, which you will get on the top of previous 3 points, if the distortion will be a complex one, like gradient swirl or multi-waves, or another (which you may find yourself)

Submission via the SCM *Mercurial*

To submit this assignment (and also the homeworks seven and eight), we shall use the Source Code Management system called Mercurial. You practice the basic use of Mercurial in the labs. The detailed description of how to clone the Mercurial repository and use the local repository to submit your work (this assignment and two last homework) is provided at this URL:

`http://cs.anu.edu.au/students/comp6700/mercurial.html`

The main repository URL is

`http://cs.anu.edu.au/students/comp6700/hg/ass2_hw7-8`

Last Remarks

If you notice an error (typo, poor language, ambiguity or contradiction) in this document let me know as soon as possible, so I can correct it for everyone's benefit.

Send an email or come and see the lecturer or your tutor for consultation if you need to discuss the assignment and/or your work.

References

[1] The Wikipedia page CAPTCHA provides the basic information and links for further studies.

[2] The `captcha.net` site `http://www.captcha.net/` also provides a good introduction and gives pointers to algorithms for possible use.

[3] Just do the search `https://www.google.com/search?q=simple+captcha+algorithms` with this or similar query.

[4] The API for *JavaFX 2.2* is located at `http://docs.oracle.com/javafx/2/api/index.html`

[5] The official *JavaFX* tutorial at `http://docs.oracle.com/javase/8/javase-clienttechnologies.htm`. It covers the JavaFX which is the part of Java SE 8; it is officially known as *JavaFX 8*, but it is almost identical to *JavaFX 2* which is what we use in the course.

Alexei Khorev
`version 1.0`, May 5, 2014
`version 1.1`, May 6, 2014