# Assignment 3

The due date: 11:00pm, Wednesday, 31st of October 2017.

*This assignment is worth of 13% of the total course mark.*

## 1 Overview

In this assignment, you will implement a complete graphical **Map editor**, in a class called *MapEditor*, that permits editing of maps and simple trip planning. The map that you display will be stored in the *MapImpl* class you wrote for assignment 2. Your **Map editor** program should be able to read a map from a file, or write one to a file, using the *MapReaderWriter* class you built previously. You will also be able to plan trips via the *trip()* method in your *MapImpl* class.

## 2 Specification

**Exercise 1** *Building the GUI*

Your GUI will have a single large panel to display a representation of the map you are editing. There will be a menu-bar with two items to allow a user to control the program:

- A `[File]` menu that contains the options: `[Open...]`, `[Save as...]`, `[Append...]`, and `[Quit]`.
  - The `[Open...]` item pops up a file dialogue (*JFileChooser*) and allows the user to choose a map file to read in and display. Any existing map on the display is discarded.
  - The `[Save as...]` item pops up a file dialogue (*JFileChooser*) and allows the user to choose a file, or enter a file name, into which a representation of the map currently presented on the screen will be written.
  - The `[Append...]` item pops up a file dialogue (*JFileChooser*) and allows the user to choose a map file to read in and append to the existing diagram. Note that the default file-extension for all map files is "`.map`".
  - The `[Quit]` item allows the user to quit the program. If the user has made changes to the map that have not been saved to a file, the program should warn the user and offer the choice of proceeding or cancelling.
- An `[Edit]` menu that contains the options: `[New place]`, `[New road]`, `[Set start]`, `[Unset start]`, `[Set end]`, `[Unset end]`, and `[Delete]`. These option are described in more details below.

You should also add a keyboard shortcut for each file-menu item (see *JMenuItem.setAccelerator()*). These are:

- `[File]->[Open]: Control-O`
- `[File]->[Append]: Control-A`
- `[File]->[Save as]: Control-S`
- `[File]->[Quit]: Control-Q`

Attach a listener to the `[Quit]` item that will cause your program to quit. For the moment, attach listeners to the other menu items that simply print out an appropriate message (such as "Open selected", when the open menu item is selected). Please, make sure that your editor can be resized!

**Exercise 2** *Verify the basics*

You should build all of the above parts in a class named *MapEditor* and then verify that each part is working before proceeding. Your *MapEditor* class should contain a *main()* method so that your program can be executed. Verify that all the menu items function correctly and print out the expected messages. Verify that `[Quit]` causes the program to quit. Do not proceed until you have verified all is well!

**Exercise 3** *Read and write files*

Extend your program so that the `[Open]`, `[Append]`, and `[Save as]` menu items correctly open and read in, append, and save map files. You should use the *MapImpl* and *MapReaderWriter* classes you wrote for assignment 2 to do this job. Open a file and verify that you can successfully save it again. Read in a file, append another to it, and write out the result. Check that the result is correct.

If an exception is thrown by the *MapReaderWriter* code, your program should pop up a dialogue box (*JDialog*) that displays the error message, and then allow the user to press an `[Ok]` button to dismiss it. Read in a file that has errors and check that the error dialogue pops up and can be dismissed. After an error, verify that you can still open another file correctly. Once again, test thoroughly.

**Exercise 4** *Build the map panel*

Create a new class, named *MapPanel*, that extends *JPanel* and implements *MapListener*, and add it to the main *JFrame* of your *MapEditor*. Make sure that the layout manager for this panel is set to null, or you will experience a lot of strange behaviour later on.

*MapPanel* will need to implement the three methods of the *MapListener* interface. For the moment, each of the three methods should simply print out a message that says something like "placesChanged" when the method is called.

**Exercise 5** *Test the MapListener*

Similarly to exercise 3, test your program by reading in files. Each time the reader-writer adds a place to your *MapImpl*, you should find that the *MapListener* you have implemented gets called and prints its message. For example, if the file you open contains four places, there should be four calls to *placesChanged()* printed out. Similarly, there should be calls to *roadsChanged()* each time a road is added, and to *otherChanged()* if a *start* or *end* place is read. Note that through all these tests *MapPanel* will be blank since nothing has been added to it yet. Don not worry, that is coming soon!

**Exercise 6** *Create a PlaceIcon*

Build a new class named *PlaceIcon* that extends *JComponent* and implements *PlaceListener*. This class will be used to display a place on the screen. You will need to override the *paint-Component()* method to make the *PlaceIcon* painting a coloured square on the screen.

**Exercise 7** *Test the PlaceIcon*

The easiest way to test the *PlaceIcon* is to add some code to the constructor in the *MapPanel* class so that it adds a *PlaceIcon* to the panel at a known location (such as 0,0). When you execute the program, you should see one square that you have drawn. However, you probably will not, because you need to get quite a lot of things "right" for this to happen. You will need to do some debugging to figure out what is wrong. If all is well, add a few more *PlaceIcon* objects at other locations to check that the program can handle multiple *PlaceIcon* objects on the screen at once. Don not proceed until you have verified that all is well!

**Exercise 8** *Display places*

Modify the *placesChanged()* method in the *MapPanel* so that each time it is called it calls the underlying *MapImpl* to find out what places have changed. To do this you will need to compare the list of places in the *MapImpl* (returned by *getPlaces()*) with the list of places stored in your *MapPanel*. (You will need an instance variable to help with this.)

Build code so that whenever a new place (say *p*) is added, your program creates a new *PlaceIcon* and adds the *PlaceIcon* as a listener to the new place *p*. Arrange that the *placeChanged()* method in the place icon sets the location of the *PlaceIcon* to the coordinates of the place *p* and then calls *repaint()*.

Then add the place icon to the *MapPanel*, so it will be displayed. Note: For each place there will be a corresponding *PlaceIcon* registered as a listener of that place. The *PlaceIcon* will need to keep a pointer to its underlying Place so that it can later get information from the place (such as its location). In future, whenever the place changes, your *PlaceIcon* will receive a call to *placeChanged()* that tells it about the change.

**Exercise 9** *Verify that the PlaceIcons appear*

Once again, read in a file as in exercise 3. You already know that each time a place is added your *placesChanged()* method will be called (verified in exercise 5). Now, with the new code you have added, you should find that your program displays a new icon for each new place. Remember that the essence of testing is simplicity first. The first file you test should have just one place in it. When that works correctly, you can try more complex files!

**Exercise 10** *Add mouse actions*

Modify the *PlaceIcon* class so that it adds a *MouseListener* to the *JComponent*. You will need to implement five methods: *mouseEnetered()*, *mouseExited()*, *mousePressed()*, *mouseClicked()*, and *mouseReleased()*. Simply make each body printing out a message that says something trivial, such as "mousePressed".

Compile and execute your program. Verify that when the mouse is pressed, clicked, released, etc. within the *PlaceIcon*, the appropriate message is printed. Do not proceed until you have verified all is well!

**Exercise 11** *Select PlaceIcons*

Modify the *PlaceIcon* code so that it includes a boolean field named *isSelected*. Modify the *mouseListener* code in *PlaceIcon* so that when the mouse is clicked on an icon, the selected field of the *PlaceIcon* is flipped from *false* to *true* or from *true* to *false*. Clicking twice on the same place should first select and then deselect it. Change the *paintComponent()* method in the *PlaceIcon* so that you can see the state of this variable by painting a filled square when the place is selected, and an outline square when it is not.

Read in a map containing several places. Click the mouse on the places and verify that you are able to select and deselect places, and that each time you do so the icon changes. (Hint: If this does not happen, try resizing the window a little bit. If resizing makes the display show correctly, then your program is missing a call to repaint.) Do not proceed until you have verified all is well!

**Exercise 12** *Complete the place-selection code*

Modify the code in *MapPanel* so that clicking on it deselects any places that have been already selected. Clicking on the background (the *MapPanel*) should clear all selections. Test thoroughly!

**Exercise 13** *Set/Unset start and end places*

Add an item to the `[Edit]` menu that allows a user to set start place. To use this option, the user first selects a place (by clicking on it) and then chooses the menu option `[Set start]`. The program should set the selected place as the start place by calling *setStartPlace()* in *MapImpl*. If more than one place is selected, pop up a dialogue that says "only one place can be selected", waits for the user to click `[Ok]`, and then ignores the operation. If you have not done it earlier,

you may need to add some code to the *PlaceIcon* object's *paintComponent()* method to display a start place in some visibly-different way (e.g. a different color or a thicker border).

If your *MapImpl* works correctly, you will not need to do anything to make the display update because the *PlaceListener* for the selected place will automatically notify the corresponding *PlaceIcon*. If this does not happen, think carefully about what needs to be changed. (Hint: You may need to modify your *MapImpl*.)

Add a menu item `[Set end]` place, that behaves similarly to `[Set start]` place. Add a menu item `[Unset start]` place that unsets the start place. Add a menu item `[Unset end]` place that unsets the end place. Test thoroughly!

**Exercise 14** *Add mouse-motion actions*

Modify the *PlaceIcon* class so that it adds a *MouseMotionListener* to the *JComponent*. You will need to implement two methods: *mouseMoved*, and *mouseDragged*. Initially, make these methods printing a simple text and verify that they are working. Now write code so that pressing the mouse-button, when the cursor is over a place-icon, and then dragging the mouse causes that place-icon to be dragged. You will need to work out how far the mouse has moved, and then call *moveBy()* on the *Place* attached to the *PlaceIcon*. (You will probably find you need information from the *mousePressed* method to help you do this.) You should not need to add anything to cause the place to be repainted after the move because the listener code you have built in the *Place* will (should!) automatically call the attached *PlaceIcon* and tell it about the change. If all is well, you will be able to click-drag a *PlaceIcon* on the screen and it will follow your mouse. Do not proceed until you have verified all is well!

**Exercise 15** *Add a selection box*

Modify the *MapPanel* code so that when the mouse-button is pressed over the background (not on a *PlaceIcon*), your program draws a selection-rectangle on the screen that follows the mouse pointer. If you drag the mouse right-and-down, the box will grow normally. If you move up or to the right, you will probably find that the box is not drawn on the screen because the *Rectangle* class cannot handle rectangles with negative width or height. You will need to modify your code so that selection works correctly in this case. If all is well, when you press-drag on the *MapPanel* background, a box will appear and follow your mouse. When you release the mouse, the rectangle should disappear.

Note that you should be able to press-drag in any direction, and the selection box should always be drawn. Hint: By now your code for handling the mouse will be getting reasonably complex. You may want to consider designing and implementing an FSA to keep track of what is happening. You will certainly need it for later stages of the assignment.

**Exercise 16** *Make the selection box work*

Now when you can draw the selection box, we need to make it working correctly. Each time the *mouseDragged* method is called, check to see if the selection-box intersects any of the *PlaceIcon* objects of the *MapPanel*. You can get the list of *PlaceIcon* objects by calling *getComponents()*. You can check for intersection using the *rectangle.intersects()* method and the

*getBounds()* method of the *PlaceIcon*. If you find a *PlaceIcon* that is inside the selection rectangle, set its *isSelected* flag to *true*. If the *PlaceIcon* does not intersect the selection rectangle, set its *isSelected* flag to *false*. (If your code is correct, the icon will automatically change to show they are selected/deselected because of calls made to the *placeChanged()* listeners.) The effect of all this should be that when you drag the selection rectangle over place icons, the ones inside the rectangle are displayed as selected, and the ones outside are displayed as not selected. Test thoroughly!

**Exercise 17** *Display the roads*

Build a new class named *RoadIcon* that extends *JComponent* and implements *RoadListener*. This class will be used to display a road on the screen. *RoadIcon* and *Road* will operate very similarly to *PlaceIcon* and *Place*, so you may be able to copy quite a lot of the code. You will need to override the *paintComponent()* method to make the *RoadIcon* drawing a straight line between the two places, and to draw text that shows the name and length of the road (in the middle of the road). Your roads need to come neatly from the boundary of the square (the printed lecture notes show how to do this in a few lines of code).

Modify the *MapPanel* so that the *roadsChanged()* method adds a *RoadIcon* to the *MapPanel* whenever a *Road* is added to the underlying *MapImpl*. You can test your implementation by reading in a file that has two places and one road. The places should be displayed correctly because of the testing you have done in earlier steps. If all is well, your *RoadIcon* will correctly display the road on the screen. If it is not, displayed, or is in the wrong position, you will need to do some debugging. Do not proceed until you have verified all is well!

**Exercise 18** *Verify that roads work*

Make a small file that has three places and three roads. Load it into your editor and display it. If all is well, there will be three places and three roads showing. Now click-drag on one of the places and move it on the screen. If all is well, the place will move (because you verified this behaviour earlier), and the road will move also. If it does not, there is probably an error in your *MapImpl*, where the *Road* is not calling *roadChanged* when either of the end-points of the road changes – think carefully about how to fix this. **Important**: The right solution is very simple code, the wrong solution is very messy! Test!

**Exercise 19** *Add a new place*

Add an entry `[Add place]` to the `[Edit]` menu that allows a user to add a new place. When the user selects this menu item, a dialogue (use *JDialog*) that asks the user for the name of the place and offers `[Ok]` and `[Cancel]` options. If the user presses `[Cancel]`, the operation is cancelled. If the user presses `[Ok]`, the program attempts to create a new place with the given name, in the middle of the screen. If there is an error (e.g. the name is illegal, or the place already exists), display a dialog showing the error message, wait until the user clicks `[Ok]`, and then ignore the operation.

To create a new place, you need only to add the place to the *MapImpl* by calling *addPlace()*. The listener mechanisms that you have built and tested earlier should then automatically create the corresponding icon for the place on the screen, and display it.

Once the new place is displayed, the user can move it to the desired location on the screen using the mouse (click-drag). You can verify that the place has been added correctly by saving the modified map to a file and checking that the new place is in the file. Check carefully.

**Exercise 20** *Add a new road*

Add an entry `[Add road]` to the `[Edit]` menu that allows the user to add a new road. This works very similarly to `[Add place]`. When the user selects this menu item, a dialogue should pop up to ask the user for the name of the road and its length, and offer `[Ok]` and `[Cancel]` options. If the user presses `[Cancel]`, the operation is cancelled.

If the user presses `[Ok]`, the program changes the mouse-cursor to a cross and waits for the user to click on the start-place for the road. Once the user has chosen the start place, the program should draw a rubber-band line from the start place to the mouse cursor. (The line should follow the mouse around as the mouse is moved.) When the user clicks on the second place, the program inserts the road into the *MapImpl*. (Hint: You will probably need to build an FSA in the *MapPanel* to manage this behaviour.) If there is an error (e.g. the name is illegal or the road already exists) display a dialog showing the error message, wait until the user clicks `[Ok]`, and then ignore the operation. (Hint: Make certain that when an error occurs, your FSA ends up in the right state.)

To create a new road, you need only to add the road to the *MapImpl*. The listener mechanism that you have built and tested earlier should automatically create the icon for the road on the screen. You can verify that the road has been added correctly by saving the modified map to a file and checking that the new road is in the file.

**Exercise 21** *Update trip distance*

Your GUI must display the distance for the current trip (as calculated by your *MapImpl*). If the endpoint is not reachable from the start point, display "No route" instead of the distance. Whenever the start- or end-place changes, your program will need to recompute the distance. You will know that a change has occurred, because your *MapImpl* will (should!) call the *otherChanged()* listener. If a route exists, your *MapPanel* must highlight the roads on that route - use a brighter colour, or a thicker line. If your *MapImpl* and *RoadImpl* are correct, this should require only trivial changes to your *RoadIcon*. If it does not work, think carefully about what exactly needs to be changed.

**Exercise 22** *Delete places (BONUS)*

You do not need to complete this step to receive full marks. If you do implement it correctly, you will receive one bonus mark.

Add a menu item `[Delete]` that deletes the selected items from the map. To use this operation, the user selects one or more places on the map (by clicking on them or by using the selection rectangle), and then chooses the delete option from the menu. The program now deletes those places from the map.

You will probably need to modify the code that implements *placesChanged()* so that it correctly handles the removal of a place or a road. If you do this correctly, you should find that the rest of your program will correctly handle the changes to the display. When you delete a place, there are many things that need to be done (unhook listeners, remove the *Place* object, remove the *PlaceIcon* object, etc) – it is easy to do an incomplete job, and then weird things will happen. You may need to print out the contents of some of your data-structures to be sure you have got this right. You can verify that deletion is working correctly by saving the modified map to a file and checking that the deleted items are no longer in the file.

**Exercise 23** *Delete roads (BONUS)*

You do not need to complete this step to receive full marks. If you do implement it correctly, you will receive one bonus mark.

Extend the behaviour of the `[Delete]` menu item so that it can delete a road. To use this option, the user selects one (or more roads) and/or places on the map (by clicking on them or by using the selection rectangle), and then chooses the delete option from the menu. The program should delete those roads and/or places from the map. The implementation issues are similar to the previous step. You will find that making roads selectable is tricky because the bounding box covers a large area of the screen.

# 3 Testing and Assessment

Since the program is controlled by a GUI, it cannot be tested automatically. We will test each GUI by hand after the hand-in deadline. During testing, we will check for features in the order described in the steps above. Marks will be awarded for each feature that works correctly.

# Submission instructions for programming code

First, type the following command, all on one line (replacing `aXXXXXXX` with your username):

```
svn mkdir --parents -m "EDC"
https://version-control.adelaide.edu.au/svn/aXXXXXXX/2017/s2/edc/assignment3
```

Then, check out this directory and add your files:

```
svn co https://version-control.adelaide.edu.au/svn/aXXXXXXX/2017/s2/edc/assignment3
cd assignment3
svn add *.java
svn commit -m "assignment3 solution"
```

Next, go to the web submission system at:

https://cs.adelaide.edu.au/services/websubmission/

Navigate to *2017*, *Semester 2, Adelaide*, *Event Driven Computing*, then *Assignment 3*. Click *Make a New Submission for This Assignment* and indicate that you agree to the declaration. The script will then check whether your code compiles. You can make as many resubmissions as you like.