PROJECT 2 – EVENT-DRIVEN PROGRAMMING

OBJECTIVES

In this project, you will write a simple graphical editor for creating and editing Finite State Machines. You will use event-driven programming and the Model-View-Controller design pattern, but learning about event-driven programming is not the only goal of the project. You will:

- Learn the details of event-driven programming by building an application using the approach.
- Learn more about unit-testing.
- Develop skill in understanding and using a complex class library.
- Learn about some more design patterns.

TO DO

1. Read about Java’s Swing toolkit, used for creating graphical user interfaces. See the course web-site for recommended sources of information about Swing. In particular, read about how components work and how to create custom components in the Swing Tutorial in the section called “Performing Custom Painting”. You can also refer to the TicTacToe example posted on the course web site for guidance.

2. Be sure you understand the Model-View-Controller or Document-View design patterns. Your program design should use one of them (probably Document-View).

3. Start small: before you design to accomplish all of the requirements given below, create a simple program that follows MVC or Document-View to display a simple state machine (in this prototype, you don’t need to allow the user to create or edit the state machine). You want to make sure you have the basics of the framework down before you try to do more complex things.

4. Design your program and document the design with at least one class diagram and at least one interaction diagram. When you are working on the design, think about how the different classes will be used together to make the overall program and think about the kinds of requirements changes that might be made if this program were to be extended later into new releases. Your design must satisfy the requirements stated below in the next section.

Think carefully about what parts of the code belong in the model class(es) and what parts belong in the view.

5. Decide how to break the project into pieces that each of you can work on independent of the others. In order to make sure this will be a success, you should decide on the APIs for the different classes as a group. Don’t worry too much about getting exactly equal-sized pieces in your design.

6. Individually, implement according to your design. Try to do this independently. However, you will probably find that as you get into the details you will need to make changes to interfaces – when this happens, work together to re-work the design, before continuing on in the coding.

To avoid this problem, try to work out the details as best you can during design, before you do the coding. The more sequence diagrams you write, the more you will be able to elucidate these details. Because the inevitable will still happen, please work ahead enough so that you have more than a day at the end to integrate the different parts.
You might also find it useful to **prototype** different parts of the design to ensure that you understand how something will work. These prototypes can be **throw-away** prototypes – don’t get too attached to them.

**Note:** Please label, via comments, the authors of the different parts of the design.

**Note:** Be sure to use **packages** to separate the code, at least into model and view.

7. Individually, create JUnit unit tests for your parts of the system. This will involve creating test doubles for the parts created by others. *You can use this opportunity to ensure that you all understand the expected behavior of each part.*

**Note:** Having good unit tests that ensure that the parts do their jobs well according to the desired APIs will enable you to work truly independently from each other, so you don’t have to spend an inordinate amount of time on integration at the end.

**Requirements**

You have flexibility to make the program however you want. However, your program should, at minimum:

1. allow users to create Finite State Machines by drawing the states and connecting them with transitions.
2. allow users to name/label states and transitions.
3. allow users to mark states as start states and accept states. Accept states should be drawn with concentric circles.
4. allow users to save/load State Machines to/from files. You can choose/invent the file format, but it would be nice if it was human readable.
5. allow users to move states around using the mouse (and have the transitions stay connected).

**Deliverables**

For this project, there are two presentations in addition to the following:

- Your design diagrams (in pdf or jpg)
- Your code (with javadoc comments)
- Your tests (as JUnit tests) and a test report (in pdf) explaining, for each test:
  - The rationale for that test. What were you trying to test?
  - The results of running the test.

All of the above should be in your team’s git repository, which you should create and which you should share with me (give me Master privileges).

1. On the first presentation day, you will present, in at most 8 minutes, a brief introduction to the interesting parts of your user interface and a preliminary version of your high-level design, paying particular attention to the flexibility of your design – explain some of the changes that your design enables. **Use patterns language** where appropriate.

**This design should be as fleshed out as possible.** You should aim for it to be a good, **final** design – though you should be willing to change it based on feedback from the presentation. If you haven’t started coding to this new design, you will be making the final week of the assignment harder than neccessary.
Feel free to show prototypes you have developed to learn how things work or to work out design decisions.

In this presentation, demonstrate a working system that satisfies some of the requirements, but is designed to make it easy to extend to the other requirements.

2. On the second presentation day, you will present, in at most 8 minutes, your final design as implemented. You should also run the program to show that it satisfies the requirements (but the focus is on the software design).

Both of these presentations should focus on design, not implementation. Show UML class and sequence diagrams and discuss the design choices you have made. Only show code if it helps to explain the design.

A Note on Diagrams. For the presentations, create diagrams that help you to focus on the parts of the design you want to discuss. Do not simply create one large diagram with all of the details and then expect to pan and zoom to show us the interesting parts. Also, bring your diagrams in electronic form, so they can be presented on screen – we will not have time for you to draw them on the board.

**EVALUATION**

I will evaluate your design primarily. A good design with a less-than-stellar implementation is better than an inflexible design with fantastic implementation. Repeat: the design is more important than the implementation.

Of course, you should aim to have the system working, but please focus your energies on making a good design. Good, flexible design, and good unit tests, will make it easier to get the system working anyway.

I will also evaluate how well you are following the suggested process and using your tools effectively. Are you using git regularly so there is a clear record of what you’ve developed so far? Are the design diagrams created and used to enable communication? Are your tests effective?

**DUE DATES**

<table>
<thead>
<tr>
<th>Step</th>
<th>Due Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project assigned</td>
<td>t</td>
</tr>
<tr>
<td>Initial Design/Presentation/Implementation</td>
<td>t + 9 days</td>
</tr>
<tr>
<td>Final Design/Presentation/Implementation</td>
<td>t + 16 days</td>
</tr>
</tbody>
</table>