Lab #2 – Virtual Pests

Total Points: 50

**Introduction:**

The most popular toy to hit the scene a few years ago was the "virtual pet" - an electronic toy that simulated pet behavior (displayed on a screen) and allowed its "owner" to respond to various types of behavior by performing actions (generally pushing buttons). Owners often find that these toys are more "virtual pests" than "virtual pets". Jeffrey L. Popyack at Drexel University has adapted this idea into a nifty programming assignment, as described below.

The behavior of a virtual pest can be described using a *finite state machine*. For example, consider a simple virtual fish that is perpetually in one of four *states*: **hungry**, **sleepy**, **full** or **grouchy**. The fish indicates the state that it is in by making a particular sound and performing a particular action. The fish also interacts with its owner by changing its state in response to owner actions. This is shown in the virtual pest window (see Figure 1), where owner actions appear as buttons.

A screenshot of a fish

Description automatically generated

**Figure 1:** Fred the fish as a virtual pest.

The state of the fish will change depending on the actions of its owner or just the passing of time. If a fish is **hungry**, and its owner feeds it, it becomes **full**. Also, if a **hungry** fish is not fed, after some time interval it will become **sleepy**, and so on. The diagram below (Figure 2) describes this behavior as a finite state machine. The machine has four states, which are drawn as circles labeled with the state's name. The machine is always in exactly one of these states. The machine may make a *transition* from one state to another as specified by the arrows connecting the states. Transitions are triggered by the actions of the pest owner or just the passing of time. For example, the arrow labeled **feed** connecting the **hungry** state to the **full** state means that feeding a **hungry** fish causes it to become **full**. The label **random** means "after some (random) time interval". The arrow entering the **hungry** state from nowhere means that **hungry** is the *initial state* - the state that a fish starts in.

![A picture containing photo

Description automatically generated]()

**Figure 2:** A finite state machine for the behavior of Fred the fish.

The virtual pests available commercially generally have very many states and there are many ways in which their owners can interact with them. They also sometimes have *terminal states*, that is, states from which the pest cannot exit because no arrows leave the state. Typically, the pest is said to have died if it enters such a state.

**Part A: Implementing the Pet Fish**

For the first part of this assignment, you will implement a pet fish with the behavior described above. That is, you will implement the finite state machine given in Figure 2. To simplify the implementation, we will assign an integer to each state and each owner action. These numbers will be used internally in your program to represent the state and action. We will also write methods that map these numbers to meaningful state and action names to display to the user.

For example, the states and actions for Fred the fish can be mapped to integers as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **state** | **number** | | hungry | 1 | | sleepy | 2 | | full | 3 | | grouchy | 4 | |  | |  |  | | --- | --- | | **action** | **number** | | random | 0 | | feed | 1 | | pet | 2 | | put to bed | 3 | |

The finite state machine can be written with these numbers for the states and transitions as follows in Figure 3.

![A picture containing photo, hanging, indoor

Description automatically generated]()

**Figure 3:** The finite state machine from Figure 2 using numbers for state and action names.

In each state, Fred makes a characteristic sound and performs a habitual action. These sounds and actions are given in the following table. In state 4, there are two spaces between "blurp!" and "bubble bubble ..."

|  |  |  |
| --- | --- | --- |
| **state** | **pest sound** | **pest action** |
| 1 | blurp! | swim to top |
| 2 | zzZZzzzZz | lay on bottom |
| 3 | BLURP! | sulk on bottom |
| 4 | blurp! bubble bubble ... | swim to bottom |

**To begin implementing the behavior of Fred the fish, complete the following:**

1. Create your GitHub repository and import the code into Eclipse as described on the "Howto…" webpage, available from the course homepage.

2. In Package Explorer, expand the lab2a/src/demoPest package. Some JUnit tests have been provided for you in this package, but the project does not yet include the JUnit library. Therefore, you will see some compilation errors in the VirtualPestTest.java file, until the JUnit library is added.

3. In the course "How to…" document (on course homepage), follow the instructions in the section titled "Adding the JUnit4 library to a project in Eclipse". Verify that the compilation errors in VirtualPestTest.java have now disappeared.

4. To run the VirtualPest game, run the VirtualPestGUI.java class. Fred the fish will appear, but the buttons for interacting with him will not work. Also, Fred will not make any sounds or perform any actions.

5. Double click the VirtualPest class to open it in the editor. The fields, constructor and several methods of the class are provided for you. Your task is to implement the remaining methods according to the given specifications. These methods are: getPestSound(), getPestAction() and doAction(). The pest sounds and actions for each state are given in the above tables.

**NOTE**: you are NOT allowed to use multiple return statements within one method definition. Instead, you should use a local variable to hold the return value from the method, and use one return statement at the end of method to return the value of that variable.

The doAction() method must implement the finite state machine given in Figure 3. This method takes an action as a parameter, and changes the state of the virtual pest depending on the combination of this action and the current state of the pest, exactly as specified by the finite state machine. For example, if the current state of the pest is **hungry** (state 1) and the owner feeds the pest (action 1), then the state of the pest becomes **full** (state 3). In addition to the owner actions, the graphical interface will also periodically pass the random action (action 0) to this method. In each state, the pest will typically only respond to some of the possible owner actions (or random) - other actions will not change the pest's state.

You must also provide test cases that achieve statement coverage for the three methods that you define. Test cases for the other methods are included with the project. Note that the constructor for the VirtualPest class takes a state number as a parameter - this is very useful for writing test cases that depend on the pest being in a particular state.

Once you have completed the implementation of the VirtualPest class, run the VirtualPestGUI.java class again. Now you can see Fred the fish in action!

Don't forget to back up your work by pushing your changes to GitHub before proceeding. Consult the How-to document for instructions on how to commit and push to GitHub.

**Part B: Designing Your Own Pest**

For the second part of this assignment, you must design and implement your own virtual pest. Navigate to the skeleton code for Part B by opening the lab2b/src/ownPest package in your lab-02 Eclipse project.

Your pest must have 4 new states, and must respond to 4 actions (including the **random** action). In the design process, you need to do all of the following:

1. Choose the states that your pest can be in, and assign a number to each state.
2. Decide what sound your pest makes and how it acts while in each state. For example, when it is sleepy, it may snore. When it is hungry, it may beg.
3. Choose the owner actions that your pest responds to, and assign a number to each action. These should be interactions of the owner with the pest, such as petting, feeding, ... Note that the number for the **random** action must be 0, and the remaining actions must be numbered consecutively starting with 1.
4. Decide how the pest reacts to owner actions (and the **random** action) while in each state, and record these decisions in a finite state machine. That is, your finite state machine should depict which states may be reached from which other states, and under what conditions, as in the example in Figure 2. You must include at least one **random** transition. Label each state with both the state name and state number, and each transition with both the action name and action number.

Note that your pest does NOT need to respond to every owner action in every state - that is, you should not have a transition from every state to every other state.

The information from points 1 - 3 above must be recorded in tables (similar to the tables given above), and you must draw a finite state machine for point 4. These tables and the finite state machine must be submitted to Moodle as part of your lab report. If desired, the finite state machine can be drawn on paper by hand; then take a photo of it and paste it into your lab report. Hand-drawn diagrams will not be penalized in any way, as long as they are legible.

Finally, implement your virtual pest. Expand the ownPest package and use the OwnVirtualPest class in this project to implement your pest, according to the given specifications.

The getNumActions(), getAction() and getFile() methods are used for building the graphical interface for the virtual pest. See the code you wrote for Part A for examples of implementing these methods. The getFile() method should simply return a the file path including the file name (as a String). This file must be a JPG image file (containing a picture of the virtual pest), and the file must be located in the lab2b/src/ownPest directory.

The getNumActions() method should simply return the number of owner actions that the pest can respond to (NOT including the **random** action, since all pests can respond to **random**). For example, if the pest responds to feeding, petting and being put to bed, getNumActions() should return 3. The getAction() method takes a number as a parameter, and returns the owner action (as a String) corresponding to that number. The numbers should range from 1 up to the number returned by getNumActions(). For example, getAction() might return "feed" when passed 1, "pet" when passed 2, and "put to bed" when passed 3. For any other number, it should return the empty string "". The getState() method returns the state that the pest is currently in as a String, or "error" if the pest is in a nonexistent state.

As with Part A, you are not allowed to use multiple return statements within one method definition for this project.

You must also provide test cases that achieve statement coverage for all of the methods and constructors in the OwnVirtualPest class (including the method and constructor that are provided for you).

In addition to your implementation, you must turn in the finite state machine and tables for your pest as specified above. The source code implementation should be pushed to GitHub and the write-up should be submitted to Moodle.

**Submitting Your Solution:**

Turn in your solution by pushing your modifications to GitHub as described in the **How-To** Document for the course. Also submit your lab report which must, as always, include a self-assessment report.

**Lab Grading Criteria:**

Your lab will be graded based on the **correctness, code quality, and test quality**. A more detailed explanation for each of these categories can be found on the “grading criteria” page linked from the Labs webpage.