UNIVERSITY OF ANTWERP
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Software Testing Lab

Assignment 5

Submission Deadline: **March 27th, 20:00**

Refer to Assignment 1 for introductory information.
Note that this assignment has a 40 point late fee instead of additional exercises.

STATE MACHINES

Now that we have monsters that can move, we should adjust the Engine such that the monsters are indeed activated. We first create a regression test suite for the engine as it exists at the moment (i.e., without monster moves), and then extend it.

- **Exercise 1.** Analyze the description of the state machine from `Engine` class as discussed in the design document (`doc/pacman-design.txt`). Create a UML state diagram from this description. Note that the actual implementation of the state machine can be inspected in `Engine` class itself. **(Required, 10 points)**

- **Exercise 2.** Make a state-event transition table (see Forgács Table 6.1, given Figure 6.2 as the state diagram, Table 6.3 is the resulting state-event table). **(Required, 5 points)**

- **Exercise 3.** Make a state-state transition table (see Forgács Table 6.2). **(Required, 5 points)**

Table 6.1 State–event notation for a state transition table

	event$_1$	**...**	**event$_N$**
start state	**next state / action**	**...**	**next state / action**
s_1	s_i / a_k	...	–
...	...	...	...
s_n	s_j / a_l	...	s_p / a_q

Table 6.2 State–state notation for a state transition table

	s_1	**...**	**s_n**
states	**event / action**	**...**	**event / action**
s_1	e_i / a_k	...	–
...	...	...	...
s_n	e_j / a_l	...	e_p / a_q

Figure 6.2 State diagram for the 'RoboDog' example

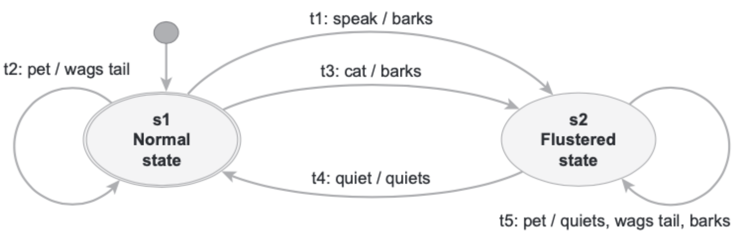

Table 6.3 State–event notation for the 'RoboDog' example with undefined transitions

	input: speak	**input: quiet**	**input: pet**	**input: cat**
start state	end state / action	end state / action	end state / action	end state / action
normal	flustered / barks	–	normal / wags tail	flustered / barks
flustered	–	normal / quiets	flustered / quiets, wags tail, barks	–

- **Exercise 4.** Create and describe a series of test case specifications that achieves transition coverage. Implement the test cases in JUnit in `EngineTest` class. **(Required, 10 points)**

- **Exercise 5.** Finite state machine specifications are often incomplete. Omitted (state, stimulus) pairs are usually ignored. Identify test case specifications for all omitted pairs, and verify that these are indeed ignored, i.e., that the state machine does not secretly implement any sneak path.Show which events are allowed or not allowed in which state. Create a sneak path test suite in JUnit to cover these scenarios. **(Required, 10 points)**

- **Exercise 6.** Analyze the coverage of your test suite, report and fix any missing test cases. Extend the models, implementation, and test suite to cater for monster moves as well. Then re-analyze the coverage. **(Required, 10 points)**

UNDO BUTTON

We will extend JPacMan with a working *undo* button. By pressing undo while playing or after just having died, the user can undo his last move, as well as any monster moves that occurred after it, after which the game will enter the halted state, from where it can press undo again. Since this is a testing course, your primary focus should be on the way in which you test this extension. In case you are running out of time, you are advised to specify test cases without working implementation, instead of an implementation without test cases.

- **Exercise 7.** Using the style of `doc/pacman-requirements.txt` document, provide a use case capturing the undo requirement. Think of possible situations that can occur, and describe the desired behavior. Summarize your design decisions in the style of `doc/pacman-design.txt` document. **(Required, 10 points)**

- **Exercise 8.** `Move` class implements the Command pattern, which was invented to facilitate undo. Write test cases for a `Move.undo` method, and extend `Move` class accordingly. **(Required, 10 points)**

- **Exercise 9.** It is probably easiest to keep track of a stack of moves in `Game` class. Write test cases that ensure proper pushing of moves made ,and popping of moves that must be undone, and provide the underlying implementation. **(Required, 10 points)**

- **Exercise 10.** Extend the JPacMan state machine diagram, the corresponding test cases, and `Engine` implementation to cater for the new undo event. Extend the JPacMan user interface with a new undo button which activates the corresponding functionality in `Engine` class. **(Required, 10 points)**

- **Exercise 11.** Given your experience with adding an undo button to JPacMan, reflect on the design of JPacMan. Describe refactorings that you found necessary, or suggest improvements to the design and code base. **(Required, 10 points)**

Late fee: -40 points