HOMEWORK ASSIGNMENT: CHAPTER 15

Question 15.3:
Derive an equivalent and one non-equivalent scenario for one of the test cases TC_A .. TC_E from Table 15.1

Per the book, the code should be (or not) in the same state as after executing:

selectModel(M1)
addComponent(S1,C1)
addComponent(S2,C2)
isLegalConfiguration()

Thus an equivalent scenario might be:

1. *selectModel(M2)*
2. *addComponent(S1,C1)*
3. *removeComponent(S1)*
4. *addComponent(S1,C1)*
5. *addComponent(S2,C2)*
6. *isLegalConfiguration()*

Further explanation about this scenario: I used a different <Model> object but the internal state of the class(es) after these method calls should be the same (..just not on a per *bit*, ie: equals(), basis)

A non-equivalent scenario might be:

1. *selectModel(M1)*
2. *addComponent(S1,C1)*
3. *isLegalConfiguration()*
4. *addComponent(S2,C2)*

Explanation: This scenario uses the same objects as parameters as the original test case and the same methods in the same sequence but will be non-equivalent after execution due to the program exiting with a different state, after the call at line 3 causes an immediate transition/exit from the states described in the statechart specification in Figure 15.6.

Question: 15.4:
Imagine we are using the equivalence scenarios approach to test a hash table class. Why might we want a toString method that returns a canonical representation of the table? Give an example of a test case in which you might use it.

Answer:

Depending on the hashing technique by which inserted items obtain their ordering – and particularly if the inserted objects require any methods for comparison that are not built-ins or available in the architecture the program runs on, natively – it is possible, if not likely, the comparison-making technique implemented in the hashing algorithm might modify the state of the program. If the state of the program is in some way determined by the hashing algorithm itself, as opposed to merely the calls before and after it, the ordering within, size of, and other factors concerning the hash table might affect the program's state. Thus, (these) **other factors concerning the hash table might lead to a non-equivalent state despite otherwise equivalent scenarios**.

An easy illustration might be two scenarios where objects of type <Group> are inserted into a hash table, but dependent upon the order in which they are inserted, the state of the program is different, despite the objects inserted being the same, otherwise.

To illustrate when this could happen let's examine an algorithm that could lead to such a scenario..

Say the hashing algorithm dictated: iff the first <Group> inserted has <Group.name.getChar(0)> <= second <Group.name.getChar(0)>, the first <Group> will have its <Group.displayName.getChar(0)> changed to the last-most available letter in the alphabet not previously used by another Group.name.getChar(0).

If we inserted new <Group>'s A,B alphabetically, Group A would have its displayName changed to, say "Z", and if we reversed the insertion order, Group A and Group B's <Group.displayName>'s would not be changed, anywhere. Thus, the insertion order could affect the state of the program, if any calls affecting the program's state check against the <displayName>'s.

In this case, a toString() method that returns the contents of the Hash Table sorted (e.g.: by <Group.name> vs. <Group.displayName>) would be useful to distinguish whether the objects inserted are equivalent (vs. relying on their "name" property – which could be misleading).