4 Lab 3: Enforcing Proper Assocations with Asserts

Binder distinguishes between implementation-based and responsibility-based assertions. The former category is used to define and check implementation assumptions. On p818, a list of implementation-based assertion types is presented.

Exercise 19 *Look up examples of the assertion types in the JPacman code. In absence, where would you introduce new instances? Against which erroneous scenarios do they protect?*

For the responsibility-driven assertions, we look at associations. Implementing associations correctly – disallowing the creation of inconsistent states – is tricky. In particularly two-way assocations are notoriously dangerous, since it is all to easy that object `o1` points to `o2`, but not the other way around. As an example, the (naive) implementation of the `CellGuest` association offered is correct, but can easily be misused (e.g. for `Cell c` and `Guest g`, `c.setGuest(g)` can be invoked irrespective of whether `g` points to `c` or to some other cell.). Recall from the class diagram (see the lecture notes) that both ends of the association have multiplicity 0...1. A simple consistency criterion should be that whenever a `Cell` thinks that it is occupied by a `Guest`, that the `Guest` knows it is occupying that `Cell`. Here we will see how to add Java assert statements to ensure to that incorrect associations are not possible (they will lead to an assertion failure), and we will see how to test such associations.

After each change, rerun the test suite with assertion checking enabled to see the effect.

Exercise 20 *Analyze the class invariants in the `Cell` class, and replace the dummy implementation (which always returns true) by a proper one.*

Exercise 21 *Do the same for the class invariant of the `Guest` class.*

Next, we will give one method only full responsibility for setting the association properly, under well-defined circumstances only. `Guest.occupy(aCell)` is the method we use for this. First, we will assume that the Cell is actually empty, so that the move is possible. Second, we will assume that the `Guest` is not already occupying some other cell. Under those circumstances, we can safely swap the association's pointers.

Exercise 22 *Add preconditions in the form of assert statements to `Guest.occupy`, listing all assumptions about the guest's current state, as well as the assumptions about the cell to be occupied.*

Exercise 23 *Repeat the same for the postconditions of this method.*

Exercise 24 *Likewise, add pre- and postconditions to the `Guest.deoccupy()` method, if relevant. Explain your answer.*

Now that the top level contracts are explicit, we can try to make the assumptions of the helper methods explicit.

Exercise 25 *Add pre- and postconditions to `Cell.setGuest(aGuest)`.*

Exercise 26 *Add pre- and postconditions to `Cell.free()`.*

Last but not least, we will revisit the test suite covering this assocation. Since the `Guest` is in charge of the association, we'll put it in `GuestTest`. This class already exists, and provides a setup method creating two guests and two cells that can be used for testing purposes.

Exercise 27 *First, add junit test cases for the correct scenarios, such as an occupy-deoccupy-occupy sequence.*

Next, since in this case the preconditions were fairly complicated and important, we add some test cases ensuring that the methods do indeed fail when invoked with a violated precondition. See `BoardTest.testFailingBoardCreation()` for an example how you could test that failing preconditions indeed generate an assertion failure.

Exercise 28 *Add a test case for an occupy-occupy sequence, which should not be permitted, since an occupy requires a deoccupy first.*

Exercise 29 *Add a test case for the `Cell.setGuest(aGuest)` method, which cannot be simply invoked for a given cell and guest.*