UNIVERSITY OF ANTWERP
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

Software Testing Lab

Assignment 3

Submission Deadline: **March 13th, 20:00**

Refer to Assignment 1 for introductory information.
Note that this assignment has a 25 point late fee instead of additional exercises.

ENFORCING PROPER ASSOCIATIONS WITH ASSERTS

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

- **Exercise 1.** Look up examples of the assertion types in the JPacman code. Where would you introduce new instances? Against which erroneous scenarios do they protect? **(Required, 10 points)**

For the responsibility-driven assertions, we look at associations. Implementing associations correctly—disallowing the creation of inconsistent states—is tricky. In particular, two-way associations are notoriously dangerous, since it is all too 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 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 2.** Analyze the class invariants in `Cell` class, and replace the dummy implementation (which always returns true) by a proper one. **(Required, 10 points)**
- **Exercise 3.** Do the same for the class invariant of `Guest` class. **(Required, 10 points)**

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 pointers of the associations.

- **Exercise 4.** Add pre- and post-conditions 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. **(Required, 10 points)**
- **Exercise 5.** Likewise, add pre- and post-conditions to `Guest.deoccupy()` method, if relevant. Explain your answer. **(Required, 10 points)**

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

- **Exercise 6.** Add pre- and post-conditions to `Cell.setGuest(aGuest)`. **(Required, 10 points)**
- **Exercise 7.** Add pre- and post-conditions to `Cell.free()`. **(Required, 10 points)**

Last but not least, we will revisit the test suite covering this association. Since the guest is in charge of the association, we will 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 8.** Add JUnit test cases for the correct scenarios, such as an occupy-deoccupy-occupy sequence. **(Required, 10 points)**

Since in this case the preconditions are 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 on how you could test that failing preconditions indeed generate an assertion failure.

- **Exercise 9.** Add a test case for an occupy-occupy sequence, which should not be permitted, since an occupy requires a deoccupy first. **(Required, 10 points)**
- **Exercise 10.** Add a test case for `Cell.setGuest(aGuest)` method, which cannot be simply invoked for a given cell and guest. **(Required, 10 points)**

Late fee: -25 points

TABLE 17.1 Implementation-based Assertion Examples

Implementation Relationship	Example Assertion
Reasonable limits on resources	`assert(NumBuff < 256);`
Incomplete or untested code	`assert(FALSE, "foo isn't complete");`
Constraints and unenforced assumptions on message arguments	`// No more than 24 total hours in day` `assert ((taskHours + slackHours) < 25 );`
Range checking on any scalar variable	`assert (dx <= maxDx && dx > 10000);`
Bounds checking on arrays and other collections	`assert (i <= sizeOfArray);` `assert (x == array[i]);`
Membership in enumerated types or collections	`assert ( !myObject.contains(x) );`
Valid, non-null pointers	`assert (this != NULL);` `assert (anArray[i] != NULL);`
"This Should Never Happen" segment in an if, case, or switch expression	`assert (FALSE);`
Language-specific weaknesses	See Implementation
Redundant or cached values	`// Buffers in use don't exceed allocation` `assert (nBuffAlloc >= buffAllocCount)`
Server object invariants	`// foo is a server object.` `assert (foo.checkInvariant( ));`
Arithmetic relationships	`assert (x >= (a + b - c));`
Items in a collection	`assert ((next == last)||(numItems > 0));`
Structures correct	`assert (p->fred != NULL);` `assert (p->refCount != 0);`
Constraints between arguments and instance/class variables	See Invariants
Constraints and unenforced assumptions on global variables	See Invariants
Constraints among instance/class variables	See Invariants
Constraints between arguments and returned values	See Postconditions
Constraints between instance/class variables and returned values	See Postconditions
Variables that must be the same (or different) at entry and exit	See Postconditions
Valid/invalid states	See State Invariants
Contract of a stub	All or part of the contract offered by the stubbed method
Contract of an abstract superclass method	The generic requirements for all subtypes