CSC 207 Software Design
Fall 2015 — Assignment 1

Logistics

- **Due date:** Thursday 22 October 9:00pm
- **Group size:** Individual

Overview

For Assignment 1, you will implement a set of Java classes according to the UML diagrams provided in this assignment description. To help make sure you have properly translated the UML diagrams into code, we have provided a type checker with your starter code; make sure to check your code with it before the assignment due date.

Learning Goals

By the end of this assignment, you should have:

1. gained an understanding of how one specifies the design for an object oriented (OO) program using basic UML class diagrams,
2. implemented a given OO design, specified by a UML class diagram, in Java,
3. become more familiar with Java generics, interfaces, abstract classes, and inheritance,
4. practised writing high quality Java documentation.

What to do for this assignment

1. Update your local copy of your subversion repository to get the starter files for the assignment. You should see a directory called `a1`. It contains a directory `A1soln` with a subdirectory `src`, which in turn contains the file `TypeChecker.java` and subdirectories `game`, `sprites`, and `ui`.
2. Start Eclipse and select the directory `a1` from your local copy of your subversion repo as a workspace to work in throughout this assignment.
3. Create a new Java Project called `A1soln`. You should now be able to view the starter packages.
4. Complete the Java program according to the UML diagram in Figures 2 through 4. Commit often!
5. Run `TypeChecker.java` by clicking on it and selecting "Run As..." ⇒ "Java Application" from the Eclipse "Run" menu. When the type checker reports that

   ```
   All type checks passed!
   Note that this does NOT mean your code works perfectly.
   ```

   your code has passed the type checker tests. Passing the type checker tests is a necessary but not sufficient condition to pass our test suite to check your program correctness (i.e., if you fail the type checker tests, you will fail the test suite but passing the type checker does not mean you will pass all our tests).

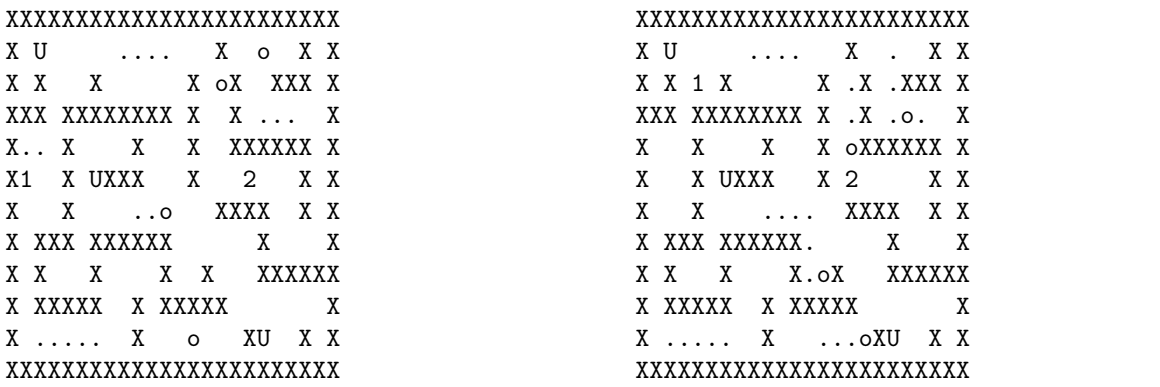

Figure 1: Example Initial and Intermediate States of the Game

6. To submit your work, add any newly created Java files and commit all your changes under the existing directory `A1soln`.

   Commit only `.java` files. Do **not** commit `.class` files or the files and directories generated by Eclipse, such as `bin`, `doc`, `.project`, etc. Marks will be deducted for submitting these files.

The Great Vacuum Race

We are implementing a simple game. The game board is a 2-dimensional grid. Each cell in this grid represents either a section of hallway, a piece of wall, or a dumpster. We denote a wall with the symbol `X`, a hallway with a blank space, and dumpster with `U`. There are two players in this game — two vacuums. We denote them with symbols `1` and `2`. Some of the cells are dirty: they contain dirt (`.`) or dustballs (`o`). The vacuums' objective is to clean up as many dirty cells as possible. The dirt is stationary, but the dustballs move about the grid and each cell that a dustball visits becomes dirty (if it wasn't already) when the dustball leaves. Figure 1 (left) shows an example initial state of the vacuum game. Notice the two vacuums (`1` and `2`), 16 dirty cells (`.`), four dustballs (`o`), and three dumpsters (`U`).

In our implementation, vacuum `1` moves left when the user presses the `'a'` key, moves right on the `'d'` key, moves up on the `'w'` key and moves down on the `'s'` key. Similarly, vacuum `2` moves left on the `'j'` key, moves right on the `'l'` key, moves up on the `'i'` key and moves down on the `'k'` key.

Only one vacuum can move at a time, but they do not need to alternate turns (e.g., vacuum `1` could move three times in a row). A vacuum cannot move onto the other vacuum (or a wall, of course), but can move onto dirt, a dust ball, a dumpster (or clean hallway, of course). After a vacuum moves, if there are dustballs, they move randomly.

When a vacuum enters a cell, it cleans the cell – i.e., any dirt or dustball is removed; the cell becomes a clean hallway. Figure 1 (right) shows an example state of a vacuum game after vacuums `1` and `2` made some moves. Notice that the dustballs moved (dirtying more hallways) and that vacuum `1` cleaned some dirt!

Each time a vacuum cleans dirt, the vacuum's score is incremented. In our implementation, dustballs are worth more. The score accrued for cleaning dirt or a dustball is defined in the starter code.

Each vacuum has a capacity. Cleaning up dirt or a dustball adds a constant amount to the fullness of the vacuum. When a vacuum becomes full, it cannot clean any more dirt; it can still go to dirty hallways, but it has no effect on the dirt that is there. A vacuum that enters a cell with a dumpster is emptied (i.e., it has zero fullness); thus, if a vacuum is full, in order to resume cleaning dirt, it must visit a dumpster. Dumpsters have no limit on their capacity.

The game ends when all dirt (including dustballs) is gone. The vacuum with the higher score wins, or, if the two scores are equal, we declare a tie.

The Implementation

We have largely designed The Great Vacuum Race game for you. Your task is to study the UML class diagrams provided and to produce a corresponding Java implementation. In addition to what is specified, you may add private helper methods.

`Grid<T>` is a (generic!) interface that defines a two dimensional grid of objects of type `T`. The class `ArrayGrid<T>`, which implements this interface, defines, among others, a `toString()` method, which needs to produce a `String` representation of the `Grid` in exactly the same way as specified in Figure 1. The class `ArrayGrid<T>` also defines an `equals` method, which returns `true` if and only if the given object is an `ArrayGrid<T>` with the same dimesions and contents as the caller. The UML class diagrams for `Grid<T>` and `ArrayGrid<T>` are given in Figure 2. This code belongs in package `game`.

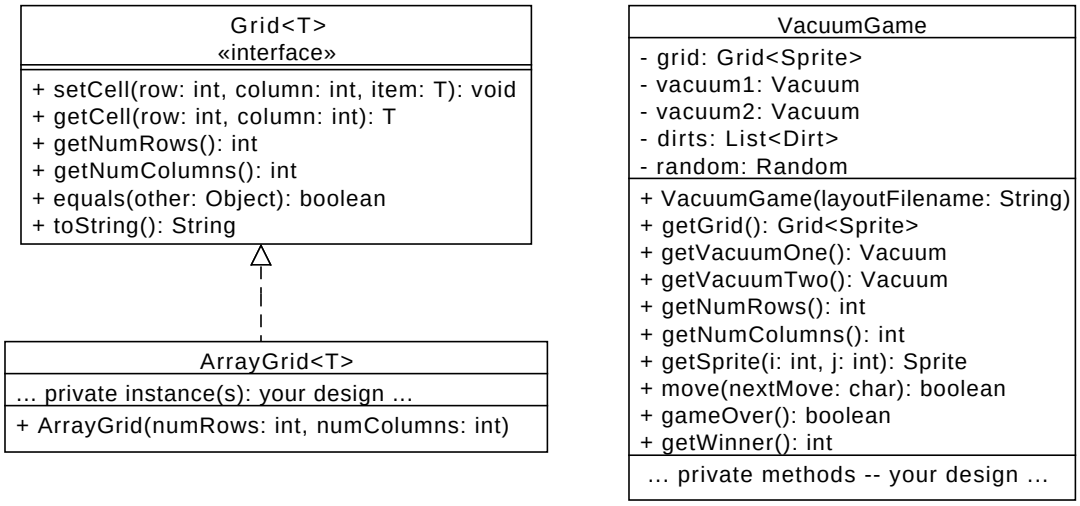

Figure 2: `Grid` and `VacuumGame` (in package `game`)

For our vacuum game, you will store and manipulate a `Grid` of `Sprite`s. Carefully study the UML diagram that defines the various `Sprite`s that will go on the grid, and the relationships between them. See Figure 3. This code belongs in package `sprites`. Here are a few things to note about the `Sprites`:

- In addition to the `Grid` keeping track of all its `Sprites`, each `Sprite` itself keeps track of where it resides.
- Each `Dirt` object stores in its `value` attribute the score accrued for cleaning it.
- Every `Vacuum` sits on top of another `Sprite`, even if it is just a `CleanHallway`. The `Vacuum` itself, not the `Grid`, stores what's `under` it. No other `Sprite` can have anything under it. Initially, a `Vacuum` sits on top of a `CleanHallway`.

The class `VacuumGame` is where all the action will happen! Class `VacuumGame` belongs in package `game`. It keeps track of the `Grid` of `Sprite`s, the two `Vacuum` players, and all the `Dirt`s. In addition to getters, this class will implement the following methods:

- The constructor for `VacuumGame` takes the path to a text file and reads and initializes the `Grid` from it. The starter code has examples that demonstrate the correct content and format for an input file.
- The method `move` takes a character. If it is a valid move character, and the move it describes is both on the grid and an allowed move (see the description above and the starter code), this method changes the position of the corresponding vacuum accordingly. Afterwards, the `Dustball`s move randomly

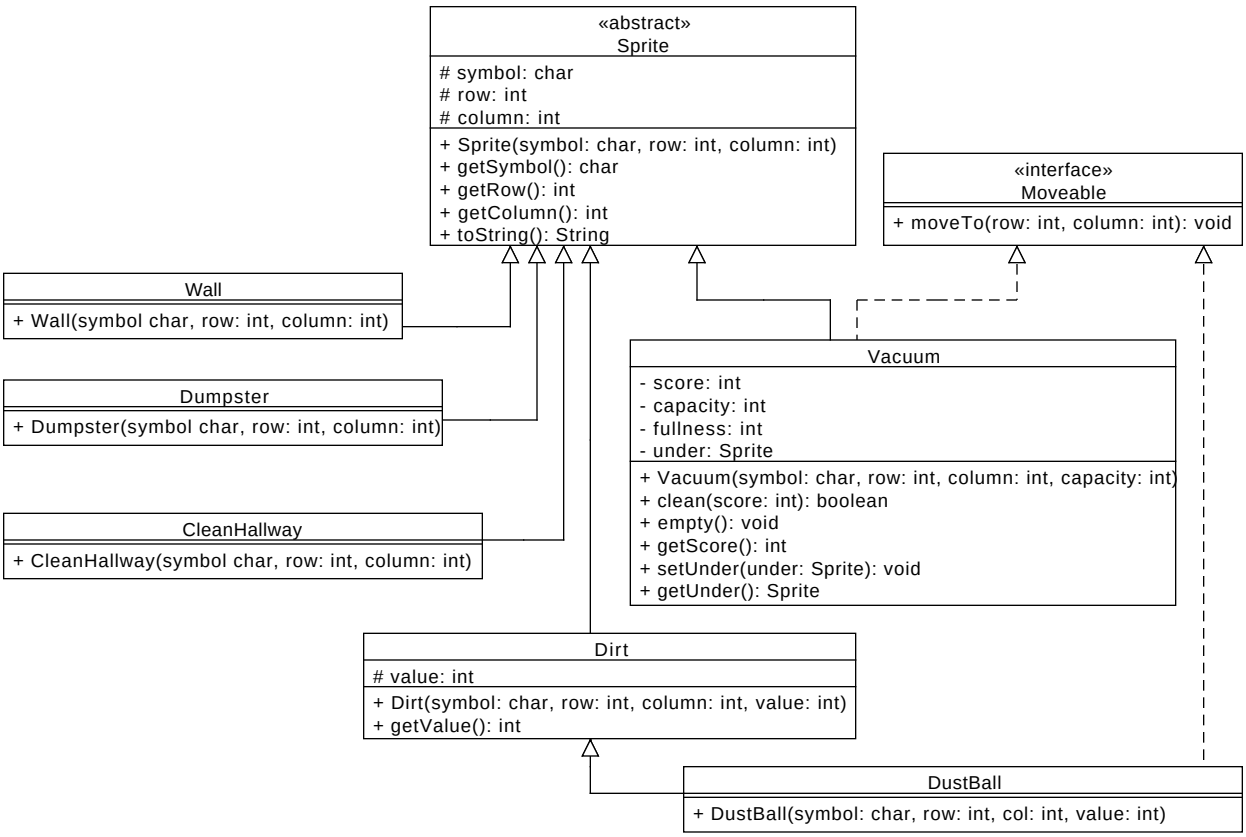

Figure 3: The Various Sprites (in package `sprites`)

(regardless of whether the vacuum actually ends up moving). `Dustball`s can only move onto cells that contain a `CleanHallway` or `Dirt`, and when a `Dustball` moves onto a cell, the `Dustball` is visible instead of whatever was there. If the input does not specify a valid move, the method does nothing. A vacuum cannot move to a cell occupied by the other vacuum. When a vacuum moves onto a dirty cell, it cleans the dirt.

- The method `gameOver()` returns `true` if the game is over and `false` otherwise.
- The method `getWinner()` returns the ID of the winning vacuum.

You will no doubt define several `private` methods (helper methods) in `VacuumGame`, in addition to the required `public` methods specified in the UML diagram.

Finally, the interface `UI` defines a user interface for the Vacuum Game. We have provided a very simple GUI implementation (see class `GUI.java`, which uses class `GUIListener.java`). Your task is to provide a text UI that implements the same interface. The text UI should read from `System.in` and print to `System.out`. See Figure 4 for the UML class diagrams. This code belongs in package `ui`.

The Starter Code

Your repository contains the starter code for this assignment. Make sure to study it carefully before you begin coding! Pay careful attention to the class `Constants`: you should make extensive use of it in your implementation.

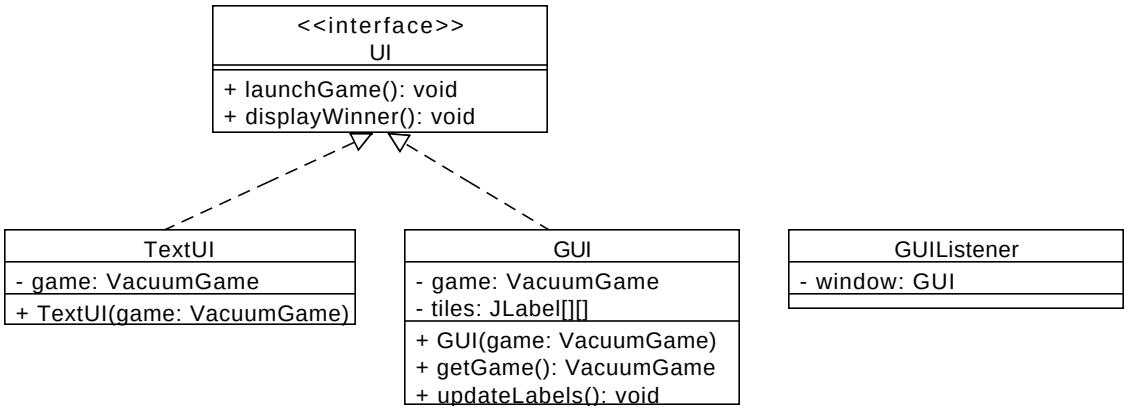

Figure 4: The User Interface (in package `ui`)

Evaluation

In addition to correctness, your submission will also be marked for style. Please follow the style guidelines outlined in `style.pdf`.

Checklist

Have you...

- followed the style guidelines?
- documented your code well? Generated the HTML documentation to see it in the form that the TAs will grade?
- tested your code on the lab computers?
- committed all of your files (and no extra files) in the correct directory?
- used `svn list` and `svn status` to verify that your changes were committed?