**Clean Code Workshop**

**Introduction**

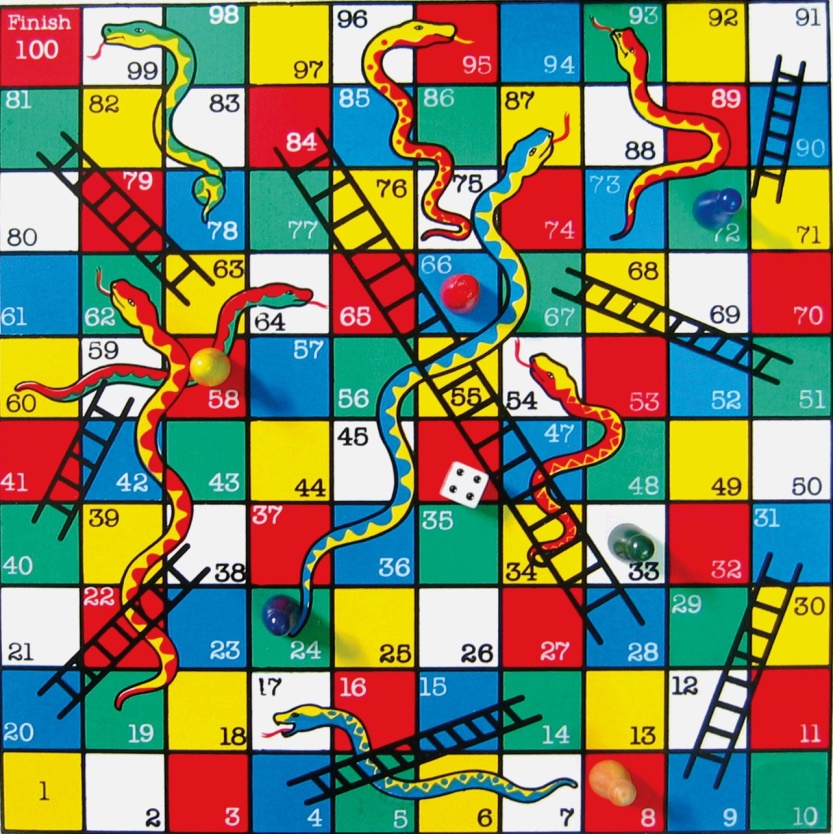
In this workshop, students will be organised into small groups and will be presented with a brief problem statement. Students will be tasked with implementing the system according to the principles of clean coding as discussed in the Clean Coding course.

The assignment will be in C++. Students will be expected to produce a working system that illustrates sound understanding of the following:

* How to split a task and allocate work to team members
* Appropriate use of classes, associations, and inheritance
* Use of design patterns, as appropriate
* Meaningful names for all code artefacts
* Quality of implementation of code and algorithms
* Appropriate use of comments

**Scenario**

In this workshop you will design and implement a C++ application to simulate the Snakes and Ladders game. A Snakes and Ladders game has 100 squares, and it looks something like this:



Here's how the real Snakes and Ladders game works.

* There can be up to 4 players (let's say). Each player has a different colour counter, e.g. red, green, yellow, or blue.
* Each player starts at square 1 and finishes at square 100.
* Each player takes it in turn to throw a dice. The dice gives a random number each time (1 to 6). The player advances by the number of the dice.
* Some squares are connected by snakes (e.g. squares 17 and 7 in the picture above). If a player lands on square 17, they slide down the snake back to square 7 in this example.
* Some squares are connected by ladders (e.g. squares 4 and 14 in the picture above). If a player lands on square 4, they climb up the ladder to square 14 in this example.
* To win a game, a player must land exactly on square 100. If a player throws too high a number, they go to 100 and then "bounce back" the left-over amount. For example, if a player is on square 98 and throws a 5, they'll go forward 2 squares to 100 and then bounce back 3 places to finish at square 97.
* The game is over when the first player gets to square 100.

Here are some guidelines, to help you plan how you're going to write your application:

* At the start of a game, the application must compute a random set of snakes and ladders. There should be 4 snakes and 4 ladders in each game. Randomizing their position will give the user hours of fun. One snake/ladder should be big, the other 3 should be relatively small.
* There must not be a ladder or snake starting at square 1 or 100. Think about it...
* At the start of a game, the application must ask how many players want to play this time, and ask each user what colour they want to be. Make sure each player chooses a different colour.
* Before each player's turn, the application should tell the player it's their turn (e.g. Peter, it's your turn now). Also, remind the player where they are, and also tell the player what snakes and/or ladders lie ahead in the next 6 squares. This gives the player a sense of excitement as they prepare to roll the dice :-)
* After a player's turn, the application should tell the player where they are now, and whether they hit a snake or ladder.
* At the end of the game, ask if the user wants to play again. If so, start over afresh.

**How to tackle this assignment**

**Planning**

* Divide into groups of 3 or 4 students.
* Start off thinking about what classes you might need. A good way to identity candidate classes is to read through the requirements stated above, and underline all the nouns or noun phrases (e.g. Board, Player, Snake, Ladder, Dice, etc.). Each of these is a potential class in the application.
* Write a UML class diagram to show how these classes might relate to each other, and decide what operations are required in each class.
* Write some UML sequence diagrams as well, where you think they will be helpful, to try to get a more detailed understanding over how objects interact over time. For example:
  + The Board object sends a message to the Dice object to generate a random number.
  + The Board object then tells the Player object to advance the specified number of squares.
  + Etc.
* When you think you understand what's needed, discuss the plan with the instructor. Describe what classes you think are needed, what behaviour they'll exhibit, and any other important decisions you've reached.

**Implementation**

* When you're ready, start implementing the classes. We suggest you split this work between the group - i.e. one student might implement the Dice and Player classes, another student might implement the Board class, etc. Make sure you divide the work up evenly.
* Make sure your classes are compatible with what everyone else is doing in your group. You'll need to plan for this – how are you going to make this work in practice?

**Completing the assignment**

* Complete the implementation of your classes, and integrate everyone's work into a single working application. Don't underestimate how long it will take to get the whole application working smoothly.
* Test the application thoroughly against the requirements.
* Demonstrate the application to other students. Describe its features, point out any nice bits, and explain if there were any bits you didn't manage to finish (that's ok, real software development projects often have to scale back on functionality if time is tight).