Self-Assessment – Design and Implement

# Going from Requirements to Design, from Design to Code

## **Activity 1 – Understand Requirements**

If the information below is unclear or incomplete, ask the instructor for clarification.

(You an also ask for more clarifications later. You don't need to nail down every single detail right now. But you should feel confident that you understand the problem and its scope.)

Your task is to design and build a program that lets two users play a game of backgammon. Slightly simplified rules of backgammon are provided below in Appendix A. You may also find it helpful to look online for illustrations of the board and other descriptions of the rules, or to play a game or two online (there are lots of free, no-registration sites). Note that you don't have to understand all the rules to start designing and coding – just figure out which entities you need to model, and what kinds of behaviors they have.

Half the point of this exercise is time management. I hope to see some good design work, but I also expect to see some code. Don't wait until the last hour to start coding! "The perfect is the enemy of the good": in the time available, "good enough" will have to do at every step.

This is a team exercise; you are allowed to divvy up the work! (I hope you break it up along natural "seams" with well-defined interfaces between them…)

## **Activity 2 – User-Visible Design**

Make a list of all the entities the user is aware of : dice, stones, board, etc. Consider their relationships, properties, and behaviors. Formulate some ideas of how you could model them. Try to apply the Six Strategies, Two Tips, and SOLID.

Draw appropriate diagrams and discuss them with a partner.

Prioritize which behaviors are most essential to the game; you'll want to implement those first.

## **Activity 3 – Internal Design**

List any entities that your program may need to have in order to support the user-visible entities. Add them to your design diagram(s) and discuss with a partner. Try to apply the Six Strategies, Two Tips, and SOLID.

Take a peek at the Extra Credit sections of this assignment and see if your proposed design is extensible to accommodate those extra features.

**HINT**: to simulate dice, you will need some code to generate pseudorandom numbers. Suitable code is provided in Appendix B.

## **Activity 4 – Framework and Prototype(s)**

Start implementing your design. Make sure the major pieces fit together, before you fill in all the details. If there's an important piece of your design that you're not confident in, write a prototype so you can test whether your solution is workable.

**HINT**: Think about ways to simplify gameplay and testing. Perhaps start with a shorter track, and fewer pieces, so that each game can be completed quickly. Which other features are not absolutely necessary for a testable game?

## **Activity 5 – Continuous Refinement**

Don't forget style, comments, or testing! Writing comments first provides documentation that will guide your coding and your testing.

You should never go more than a few minutes without compiling your code. Add features a little bit at a time. For example: first add the ability to make a move, then add input validation ("sorry, no such space #745, enter another space"), then check for illegal moves ("sorry, you can only move the number of spaces shown on the dice"; "sorry, you cannot move there, that space is blocked by two enemy pieces"), then add logic for handling captures and returning captured pieces to the board.

Implement all the simplified rules of Appendix A, then start on the "Extra Credit" rules, or other Extra Credit features below, or your own ideas.

## **Extra Credit Features**

Some ideas for spiffy features:

1. Show the board from each player's perspective when it's their turn, so each player is always moving their pieces from left to right, and from space "#1" past space "#24".
2. Highlight legal moves. Use some kind of visual cue to help the player see which moves are available.
3. Computer opponent. Have a single-player mode in which the player plays against the computer. (Note: having the computer play *well* is a very large challenge. Start by having the computer pick any legal move using some very simple criterion, or at random. If you have time, you can try to refine the computer's strategy. You may find that a *strategy* is an entity your design should deal with explicitly…)

## **Appendix A: Rules of Backgammon**

(Slightly simplified – not for tournament use!)

Backgammon is a racing game in which you're allowed to block your opponent's pieces or knock them off the track. The two players (Red and Black) are racing on the same track, but going in opposite directions.

**The track:**

The track is is 24 spaces long. (The spaces are called "points".) Each player has 15 pieces ("stones"). The goal is to move all your stones past one end of the track, before your opponent can move all their stones past the opposite end. If the spaces are numbered 1 to 24 from left to right, then Red attempts to move all his stones rightwards past space 24, while Black attempts to move all his stones leftwards past space 1.

Note: a traditional backgammon board is rectangular, with the track folded into a horseshoe shape.

*HINT: I suggest you depict your track as linear. It's easier to display on a console, and less confusing to play on too.*

**Movement:**

Movement is by rolling dice. On your turn, you roll two six-sided dice. First, move one of your pieces the number of spaces shown on one die, then move one of your pieces (perhaps the same piece) the number of squares shown on the other die. You may not move pieces backwards, only forwards (whichever way is "forwards" for you, i.e. rightwards if you're red, leftwards if you're black).

**Capture:**

You can capture an opponent's piece by landing on the space it occupies – but only if there is exactly one piece on that space.

**Effects of Capture:**

A captured piece is moved to a waiting area known as the "bar". If one of your pieces (or more than one) is on the "bar", you must move it (or them) before you move any pieces that are already on the board. Pieces moving off the bar start at (your) beginning of the track. For example, if Red has a piece on the bar and rolls a 2 and a 5, he may move the piece from the bar to space 2 or to space 5. If space 5 is blocked (see below), he must move the piece from the bar to space 2. He must then advance any piece by 5 spaces (if legal). His turn is then over.

**Illegal moves:**

It is illegal to move to a space occupied by two or more of the opposing player's pieces. Pieces on a point are immune from capture while there are two or more of them on that point. You may have as many pieces of your color on a point as you like.

*Game strategy: a cluster of double-occupied spaces makes an effective roadblock for your opponent's pieces. Six double-occupied points in a row make a completely impassable wall.*

If you have no legal moves, you forfeit your turn. If you can move, you must move.

**Winning:**

Advancing a piece past the end of the track is called "bearing off". Once it has been borne off, a piece is out of the game. You win by bearing off all your pieces before your opponent bears his off.

**Starting Position:**

Traditionally, the starting position is as follows:

Red (who moves from low to high numbers): Two stones on point #1, five on #12, three on #17, five on #19.

Black (who moves from high to low numbers): Two stones on point #24, five on #13, three on #8, five on #6.

ADDITIONAL RULES (EXTRA CREDIT FOR IMPLEMENTING):

1. Rolling two-of-a-kind (two sixes, two threes, etc.) requires you to move as if you had rolled four of that kind. For example, if you roll two ones, you move as if you had rolled four ones: four moves of one space each, distributed among up to four pieces. (If four legal moves are available.) If you roll two threes, you move as if you had rolled four threes.
2. Bearing off is only permitted when all your pieces are in their final six squares (except pieces already borne off). If one of your pieces is captured after you begin bearing off, you must bring it all the way back to the final quarter of the track before you can continue bearing off.
3. Backgammon is usually played repeatedly. Score one point for each game you win. (Two points if you win before the opponent gets all his pieces onto his final six spaces.)
4. At any point during a game, a player may challenge the other to double the stakes. This is called doubling. The challenged player may accept or forfeit. Example: red challenges black. If black declines, he loses the game instantly, and red scores one point. If black accepts, whoever wins the game will score two points.
5. The same player cannot double twice in a row, but the opposite player can redouble. Example: later in the same game, black challenges red. If red declines the challenge, he loses the game and black scores two points. If red accepts, the game is worth four points to whoever wins. Black is now free to re-re-double if and when he chooses.

## **Appendix B: Pseudo-Random Numbers**

WARNING: generating and using random or pseudorandom numbers can be a VERY subtle and tricky business. The code shown here is pretty good, but you should still consult an expert if attempting serious encryption or a large physics simulation.

The C++ standard library provides a pseudorandom number generator. You use it in two parts.

Unlike a random number generator, a pseudorandom number generator behaves deterministically. If you know the starting point, you can predict the entire sequence.

So, the first thing we do is set the starting point to something as unpredictable as we can.

Setting the starting point is called *seeding* the generator. For our "seed", we'll use the current time and date, in the highest-precision units our computer can provide (typically nanoseconds).

// SECTION ONE: DO THIS AT THE BEGINNING OF YOUR PROGRAM

// Create a seed for our random generator's engine

// If you don't provide a seed, you'll get the same sequence of dice rolls

// every time you play the game

// -- which can be useful for testing, sometimes,

// but is generally not what you want.

// For our seed we'll use the current time, in nanoseconds since 1970

// (or similar: units and Epoch date are system-dependent)

auto seed = static\_cast<default\_random\_engine::result\_type>(

chrono::high\_resolution\_clock::now().time\_since\_epoch().count() );

// Initialize an engine for a (pseudo-) random number generator, using our seed

auto engine = default\_random\_engine(seed);

// Make a generator, by binding our engine to a distribution.

// In this example we want the output distributed uniformly

// over the integer range [1,6]

auto gen1to6 = bind(uniform\_int\_distribution<>{1, 6}, engine);

/\* Note: you could combine the three steps above into one line of code, if you don't mind going blind \*/

//auto gen1to6 = bind(uniform\_int\_distribution<>{1, 6}, default\_random\_engine{static\_cast<default\_random\_engine::result\_type>( chrono::high\_resolution\_clock::now().time\_since\_epoch().count() )});

// END OF SECTION ONE

Once that's done, we can call on our generator whenever we want to roll a die.

// SECTION TWO: USE THE RNG WHENEVER YOU WANT

// now you can use the generator whenever you need to roll a die

cout << gen1to6() << endl; // generate an integer in range [1,6]; print it

// loop for illustration purposes

for (int i = 0; i < 10; i++)

cout << gen1to6() << endl; // print an integer in range [1,6]