STV Project 2019/20, PART 1

Deadline: see website.

The overall goal of this project is to learn how some basic concepts and techniques in software testing can be applied in practice. To simulate a real-life problem, you will start by developing an application and do unit testing on its components. We will do this project in two PARTs; this is the first one.

The software to implement is a console-based, single player turn-based game inspired by the classic rogue RPG game. The game is played in a dungeon in the form of a connected graph. The goal is to survive the dungeon, and reach its exit. Evil monsters roam the dungeon, but there are also items which can help the player to defeat them.

The implementation language for this project is C#.

A starting implementation will be given to you, though it leaves most of the game logic unimplemented:

https://git.science.uu.nl/prase101/STVrogue

You can clone it, and read its `.sln` file into your IDE. This initial implementation prescribes the architecture of the game logic. Please stick to this architecture and do not change the signature of existing methods.

The list of features to implement is kept minimum, to let you focus on the above mentioned goal. Some degree of complexity is deliberately introduced, to provide some challenges.

> I also need you **to keep track of your unit testing effort and finding** (the hours you spend on testing and the number of bugs you find).

1 Required software

You need an IDE for C# that includes a code coverage tool. There are two options:

1. Jetbrains Rider[1]. This has my preference. If you use Mac or Linux, you should use Rider. You can get free education license for this[2].
2. Microsoft Visual Studio Enterprise Edition. You need the *Enterprise* edition. It is a bit overkill, but smaller edition does not include any code coverage tool. Microsoft supposedly also offer an education license for this[3]. You can login with your Solis account. Unfortunately Microsoft also asks for your phonenumber, so if you don't like this you should go with Rider.

For Unit Testing we will be using NUnit Testing Framework[4], but your IDE should get this automatically when you read the project's `.sln` file into the IDE.

You will also need a Code Metrics tool. An important metric is the McCabe/Cyclometic metric (Wikipedia gives an adequate explanation on this). If you use Rider you need to install the CyclomaticComplexity plugin. If you use Visual Studio for Windows, code metrics are already included.

2 The Game Logic

The game logic is implemented by the classes in STVrogue.GameLogic. A large part of these classes are left unimplemented for you. And yes, you will also need to test them to make sure you deliver a correct game logic.

2.1 Class GameEntity

Monsters, items, rooms, and the player are the main entities of the game. They will have their own class, but they all inherit from a class called GameEntity. We will insist that game entities (so, instances of GameEntity) should have *unique IDs*. This is will be in particular important for PART-2 of the Project.

2.2 Class Dungeon

The game is played on a dungeon, which consists of rooms. There are three types of rooms: *start-room*, *exit-room*, and other rooms (we will call then 'ordinary' room). A dungeon should have one unique start-room and one unique exit-room.

Rooms are connected with edges. If r is a room, all rooms that are directly connected to r are called the *neighbors* of r. The player and monsters can move from rooms to rooms by traversing edges. Technically, this means that the rooms in the dungeon form a graph whose edges are bi-directional. We require that *all rooms in the dungeon are reachable from the start-node*. Figure 1 shows some example of dungeons.

The class Dungeon has two basic operations:

1. A constructor Dungeon($shape, N, \gamma$) to create a dungeon consisting of $N>2$ rooms.
   a. Keep in mind that a dungeon should satisfy the previously mentioned constraints about its connectivity and the uniqueness of its start and exit-rooms.

[1] https://www.jetbrains.com/rider/
[2] https://www.jetbrains.com/community/education/#students
[3] https://azureforeducation.microsoft.com/devtools
[4] https://nunit.org/

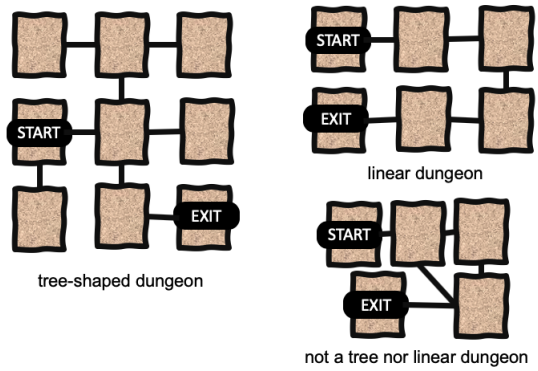

Figure 1. *Some examples of dungeons: tree-shaped (left), linear dungeon (top right), and a dungeon which is neither tree-shaped nor linear (bottom right).*

   b. The parameter *shape* determines the shape of the dungeon. There are three types of shapes: *linear*, *tree*, and *random*. A linear-shaped dungeon forms a list with the start-room at one of its ends, and the exit-room at the other end.
   A tree-shaped dungeon forms a tree with the start-room as the root. The exit-room should be a leaf of this tree.
   A *random* dungeon can has any shape as long as it is not linear nor a tree.
   c. Every room in the dungeon has a *capacity*. If c is the capacity of a room r, the number of monsters in r should not exceed r. This capacity is a random value determined when the dungeon is created. For an ordinary room, it should be in $[1..\gamma]$. Note that different rooms may thus have different capacity, but none will exceed the maximum capacity γ.
   The start and exit-rooms always have capacity 0.

2. A method SeedMonstersAndItems(M, H, R) to randomly populate the rooms in the dungeon with monsters and items. There are two types of items: healing potion and rage potion.
   The paramemer M specifies the number of monsters to be dropped in the dungeon, H is the number of healing potions to be dropped, and R is the number of rage potions.
   Populating the dungeon are subject to the requirements set below. Meeting these requirements are not always possible (e.g. it is not possible to populate a dungeon with N rooms whose capacities are at most k with $(N-2)k$ monsters or more).
   The method SeedMonstersAndItems returns true if it manages fullfill the requirements, else it returns false (and left the dungeon unpopulated).
   a. Every monster in the dungeon should be alive and have HP and AR >0.
   b. Every room cannot be populated with more monsters than its capacity allows.
   c. Let E be the set of neighbor rooms of the exit-room. So: $E = exitroom$.Neighbors. Every room in E should be populated with more monsters than any non-E room. So, for any $r \in E$ and $r' \notin E$, then $|r.\text{monsters}| > |r'.\text{monsters}|$ should hold.
   d. Let N be the number of rooms in the dungeon. At least $\lceil N/2 \rceil$ number of rooms should have no item at all.
   e. Let $I = startroom$.Neighbors $\cup \{startroom\}$. There should be at least one healing potion and one rage potion somewhere in I.

2.3 Class Creature

A creature has *hit point* (HP), attack rating, and its location (the room it is in) in a dungeon. Attack rating should be a positive integer. A creature is *alive* if and only if its HP is >0. There are two subclasses of Creature: *Monster* and *Player*.

Creature has two operations: move(r) to move it to a neigboring room, subject to the room capacity, and attack(f) to attack another creature f provided it is located in the same room. When a creature c attacks f, the action will damage f's HP (that is, reducing it) by Δ where Δ is the attacker's attack rating. If f's HP drops to 0, f dies. A base implementation of these two operations are already provided, though you will have to override it for Player, e.g. because player moves are not constrained by rooms' capacity.

The player has additionally 'Kill Point' (KP) that is increased by one each time it kills a monster. The player also has a bag, that contains items it picked up.

2.4 Items

Items are dropped in the dungeon. When the player enters a room that contains items, it can pick them. The items will then be put in the player's bag.

There are two types of items: *Healing Potion* and *Rage Potion*. A healing potion has some positive healing value. When used, it will restore the player's HP with this value, though the HP can never be healed beyond the player's HPMax.

A rage potion will turn the player into a raging barbarian. This temporarily double the player's attack rating. The effect last for 5 turns (including the turn when it is used), though it also has some consequence that will be explained later.

Using a potion will consume it.

2.5 Class Game

The class provides the entry point to the game logic, and also holds most of the game logic[5].

The constructor Game($conf$) takes a configuration and will create a populated dungeon according to the configuration. The configuration $conf$ is a record ($shape, N, \gamma, M, H, R, dif$) of 7 parameters:

1. *shape* the shape of the dungeon to generate.
2. N the number of rooms in the dungeon.
3. γ specifies the maximum rooms' capacity.
4. M is the number of monsters to generate.
5. H is the number of healing potion to generate.
6. R is the number of rage potion to generate.
7. dif is the difficulty mode of the game. There are three modes: *Newbie*-mode (easy), *Normal*-mode, and *Elite*-mode.

The constructor will generate a dungeon satisfying the parameters in $conf$. Some configurations might be hard, or, as remarked in Section 2.2, even impossible to satisfy. The constructor should throw an exception if it fails, e.g. after several attemps, to generate a dungeon that satisfies the configuration.

All game entities in the generated dungeon should have unique IDs. The player should be alive, and its HP is equal to HPMax, and >0. The player always starts at the start-room of the dungeon.

STV Rogue is a turn-based game. It means that the game moves from turn to turn, starting from turn 0, then turn 1, turn 2, etc. At a turn, every creature in the dungeon, and is still alive, makes one single action. The order is random, as long as everyone gets exacly one action. Between turns, the game state does not change.

The player wins if it manages to reach the dungeon's exit-node. It loses if it dies before reaching it.

The main API of the class Game is the method update(α). This will iterate over all creatures in the dungeon as said above. A monster can choose its action randomly; this will be explained more below. The action of the player is as specified by α (you can imagine that this α represents a choice that the actual human player indicated through the game UI).

The player is *in-combat* if it is in the same room with a monster. Likewise, a monster is in-combat if it is in the same room with the player.

There are six possible actions that a creature can do, though a monster can only do four of them:

1. DoNOTHING, it means as it says.

2. MOVE r: the creature moves to another node r. This should be a neighboring node, and furthermore this should not breach r's capacity.

   MOVE is **not** possible when the creature is in combat.

   The logic for executing this action is to be implemented in the method Game.Move(c, r), where c is the creature that moves.

3. PICKUP: this will cause the player to pick up all items in the room it is currently at. The items will the be put in the player's bag. A monster cannot do this action.

4. USE i: this will cause the player to use an item i. The item should be in its bag. The effect of using different items were explained in Section 2.4.

   The logic for executing this action is to be implemented in the method Game.UseItem(i).

5. ATTACK f: the creature attacks another creature f. This is only possible if both the attacker and defender are alive and are in the same room. Also, a monster cannot attack another monster.

   The logic for executing this action to be implemented in the method Game.Attack(c, f), where c is the attacker and f the defender.

6. FLEE: the creature flees a combat to a randomly chosen *neighboring* room. The conditions for when fleeing is allowed are a bit complicated. They are given below. When multiple conditions conflict, the condition that is listed first takes precedence (e.g. conditions c and d below may conflict; in such a situation we should follow c and ignore d).
   a. A monster cannot flee to a room if this would exceed the room's capacity.
   b. The player cannot not flee to the exit-room.
   c. In rooms neighboring to the start-room, the player can always flee.
   d. After using a healing potion the player cannot flee at the next turn. So, if the the pot is used at turn t, at turn t+1 it cannot flee (and since a creature can only do one action per turn, this implies that the player can't flee in the same turn t either). At turn t+2 will be able to flee.
   e. The player cannot flee while 'enraged'. Recall that the player enters such a state when whenever it uses a rage potion (this state lasts for 5 turns, including the turn when the potion is used). However, this restriction does not apply when the game is played in the Newbie-mode.
   If the game is played in the Elite-mode things become harder: if the player is enraged while in a room (let's call it room S) neighboring to the exit-room, the player will not be able to flee from S even after

[5] For a larger game with a more complex it would make sense to introduce more decomposition. STV Rogue is not that complex though; so, to favor simplicity I will keep most of the logic centralized in the class Game.

the rage effect has dissipated. Note that this does not mean that the player can never leave S. It can do so by using the ordinary MOVE command (which is only possible if it is no longer in combat).

The logic for executing this action is to be implemented in the method Game.Flee(c), where c is the fleeing creature.

3 The Main Class

The class STVrogue.Program is the main class (the class with the Main method) that provides the console application for the game. When you start the application, it reads the game configuration from a file (configuration is explained in Section 2.5). It shows a welcome-screen, and the game begins. At each turn the game should display at least:

1. The turn number.
2. Player information: HP and KP.
3. The id of the room the player is currently at, and those of connected rooms.
4. Ids of monsters in the room.
5. Items in the room.
6. Items in the player's bag.
7. Avaialable actions for the player.

When the player does an action, print a message to the console informing the player of the effect of this action. When a monster in the current room does an action, also print similar message. Actions of monsters in other rooms should not be echoed to the console.

When the player wins or loses, print your ending message before exiting the game.

Right now the class STVrogue.Program contains a dummy implemetation just so that you can run it. Obviously, you should replace this dummy implementation with your own.

4 Important: Controlling Random

Like in many other games, some parts of STV Rogue are required to behave randomly (e.g. when generating dungeons, or when deciding monsters' actions). When testing a program that behaves non-deterministically, the same test may yied different results when re-run with exactly the same inputs and configuration. Such a test is called 'flaky' or 'unrepeatable'. Obviously we do not want to have flaky tests. To this end, you need to make it so that you can configure your implementation of STV Rogue to switch from using normal random generators to using **pseudo random generators** when testing it[6]. Such a generator behaves deterministically when given the same seed. Check the class Utils.SomeUtils to obtain such a generator.

5 Your Tasks

Your tasks are listed below. All are mandatory, except Task 8. You should divide the work among your team members such that everyone has her/his fair share of testing. In fact, the author of a functionality should not be the only person to test the functionality due to her/his obvious bias.

1. Creature.Move(r) and Creature.Attack(f) (0.5 pt).
2. Dungeon($shape, N, \gamma$) (1 pt).
3. Dungeon.SeedMonstersAndItems(M, H, R) (1 pt).
4. Game($conf$) (1 pt).
5. Game.Flee(c) (2 pt).
6. Finishing the implementation of STV Rogue (2 pt).
7. Test the rest of the game logic (1.2 pt).
8. Optional: stronger testing of Flee(c) (1pt).
9. Report (0.3 pt).

All produced tests should deliver 100% code coverage[7] on their test target (e.g., your tests on Flee(c) should give 100% coverage on this method). If you deliver less, you have to explain the reason in your Report (e.g. because the uncovered parts are unreachable, or simply because you run out of time).

> Please document your test methods and in-code specifications/parameterized-tests. Write a comment describing what each test method tries to check. Inside the body of each in-code specification/parameterized test, write a comment explaining what correctness properties different parts of the specification try to capture.

The McCabe/Cyclometic metric of your method should give a rough indicator of the minimum number of test cases you would need to test it. The metric gives the number of 'linearly independent' control paths in the method (check Wikipedia's entry on Cyclometic complexity).

5.1 Test Creature.Move(r) and Creature.Attack(f) (0.5 pt)

To get you started in learning to do basic unit testing, test the above mentioned two methods to verify their correctness. The methods are already implemented, so you only need

[6] Well, a 'normal' random generator is typically also a pseudo random generator. It is just that its seed is not fixed, e.g. it is based on the system time. For STV Rogue, you can alternatively only use pseuodo random generators. It is not something you should do when implementing a real game, but in this project implementing real randomness is not our focus.

[7] Visual Studio tracks both line coverage and block coverage. The concept of 'block' coverage is explained in one of the lectures. Rider uses a different concept, namely statement coverage. It means that Rider can tell you which statements are covered or otherwise. This is slightly more coarse grained than block coverage. E.g. if you have a statement if($p||q$) x++, Rider can tell you whether or not you have executed the x++ in the then-branch, but it cannot tell whether you have explored all the possibilities for enabling its guard (either due to p is true, or q is true), because techically a guard is an expression rather than a statement.

```
[TestFixture]
public class Test_Remainder{
  // the tests:
  [TestCase(5,0)]
  [TestCase(5,3)]
  [TestCase(5,-3)]
  [TestCase(-5,-3)]
  ...
  // the in-code spec. for % :
  public void Spec_Remainder(int x, int y) {
      // check the method-under-test's pre-condition:
      if(y != 0) {
          // calling the method-under-test:
          int r = x % y
          // (a) check the method's post-condition: r is a correct
          // reminder if it is equal to x - d*y, where d is the
          // result of dividing x with y:
          Asssert.Istrue(r == x - (x / y) * y) ;
      }
      else {
        // (b) the method should throw this exception when its
        // pre-condition is not satisfied:
        Assert.Throws<DivideByZeroException>(x % y) ;
      }
  }
}
```

Figure 2. *An example of how to write an NUnit test through an in-code specification. Let's imagine we want to test C# remainder operator (%).*

to test them (and to fix them if you find bugs). Use NUnit Framework to write your tests.

5.2 Implement and test the constructor Dungeon($shape, N, \gamma$) (1 pt)

The intended behavior of this constructor is informally specified in Section 2.2. Implement the constructor. Then, formalize its informal specification as an **in-code specification** and then use NUnit **parameterized test** to test the method. Figure 2 shows an example of how to do this.

The class Utils.HelperPredicates contains some help predicates you might find useful.

5.3 Implementation and test Dungeon.SeedMonstersAndItems(M, H, R) (1 pt)

The intended behavior of this constructor is informally specified in Section 2.2. Implement it and write in-code specification for the method. This time, formulate the in-code specification as NUnit **Theory** and then test that Theory.

Check NUnit Documentation: https://github.com/nunit/docs/wiki/NUnit-Documentation. The entry about Theory should be listed under the category 'Attributes'. There is also an example of using Theory in the STV Rogue project itself.

5.4 Implementation and test the constructor Game($conf$) (1 pt)

The intended behavior of this constructor is informally specified in Section 2.5. Implement the constructor and write in-code specification for this method. Formulate it as a parameterized test. This time, use NUnit combinatoric testing feature to generate tests for the constructor. Be mindful that full combinatoric test may blow up to thousands of test cases. You may want to consider pair-wise testing instead.

Check the entries on 'Combinatorial' and 'Pairwise' in NUnit documentation. There are also examples of these in the STV Rogue project itself.

5.5 Implementation and test the method Game.Flee(c) (2 pt)

The intended behavior of this method is informally specified in Section 2.5. This method has the most complicated logic. Implement and test it.

5.6 Finish the Implementation of STV Rogue (2 pt)

Finish the implementation of STV Rogue to get a working game. Among other things, you will have to implement the method Game.Update(cmd) as well as finishing the main class Program.

5.7 Test the rest of the game logic (1.2 pt)

Finish the testing of the game logic (that is, of all classes in the STVrogue.GameLogic namespace).

5.8 Optional: stronger testing of Flee(c) (1pt)

The logic of the method Dungeon.Flee(c) is fairly complicaated. For such a method simple code coverage does not really reflect the adquacy of your tests as it cannot enforce path-level verification. Unfortunately there is no tool in the market that will let you do path coverage tracking. Let us try to compensate this by augmenting your existsing tests for Flee with combinatoric testing. Chapter 4 in Ammann & Offutt's book (Ch. 6 for 2nd Ed.) explains the main concepts. See also the slides from Week-2.

Identify the set of 'characteristics' on which the behavior of Flee(c) depends on. These typically include the method parameters (there is only one: c), but also other aspects that are not formally listed as a parameter, e.g. the location of c, the use of healing potion, the difficulty-mode, etc.

Then, decide how you want to partition each characteristic into 'blocks'. E.g. c can be a monster or the player (so, we would have two blocks for c). The position of c can be distinguished between: a neighbor of the start-room, a neighbor of the exit-room, or other room. The used difficulty-mode can be distinguished between: the newbie-mode, the normal-mode, or the elite-mode. And so on.

Translate your design into an NUnit combinatoric (or at least pair-wise) test using a parameterized test.

5.9 Report (0.3 pt, mandatory)

Make a report containing the items listed below.

1. The general statistics of your implementation:

N = total #classes	:	...
M = total #methods	:	...
$locs$ = total #lines of codes(*)	:	...
$locs_{avg}$ = average #lines of codes(*)	:	$locs/N$

(*) exclude comments

2. Statistics of your unit-testing effort:

Global Statistics		
N' = #classes targeted by your unit-tests	:	...
total coverage over GameLogic	:	...
T = #test cases (*)	:	...
$Tlocs$ = total #lines of codes (locs) of your unit-tests	:	...
$Tlocs_{avg}$ = average #unit-tests' locs per target class	:	$Tlocs/N'$
E = total time spent on writing tests	:	...
E_{avg} = average effort per target class	:	E/N'
total #**bugs** found by testing	:	...
Statistics of some selected targets		
Dungeon($shape$, N, γ)		
mcCabe metric	:	...
# test-cases (*)	:	...
coverage	:	...
Dungeon.SeedMonstersAndItems(M, H, R)		
mcCabe metric	:	...
# test-cases (*)	:	...
coverage	:	...
Game($conf$)		
mcCabe metric	:	...
# test-cases (*)	:	...
coverage	:	...
Game.Flee(c)		
mcCabe metric	:	...
# test-cases (*)	:	...
coverage	:	...
Game.Update(cmd)		
mcCabe metric	:	...
# test-cases (*)	:	...
coverage	:	...

(*) We will define the 'number of test-cases' as the number of tests that NUnit reports.

3. Explanation: if your coverage is below 100%, mention why you failed to get it to 100.

4. If you do the Optional Task (Section 5.8), describe your chosen set of characteristics and how they are divided into blocks. Describe your chosen approach of combinatoric testing, and how this is translated to NUnit parameterized test.

5. Specify how the work is distributed among your team members, in terms of who is doing what, and the percentage of the total team effort that each person shoulders.