STV Project 2024/25

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. In the later part of the project we will also do system-level testing.

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 (and there are also some bugs left there):

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 (feel free to add more methods and classes). **Keep your clone private!**

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 testing effort and findings** (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]. You additionally need to install the DotCover plugin.
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. Unfortunately, the Enterprise edition seems to be no longer in our free university deal. So, it is not a real option.

The project is configured to use .net 8.0 and C# 12. Please stick with this setup.

For Unit Testing we will be using NUnit Testing Framework[3], but your IDE should get this automatically when you read the project's `.sln` file into the IDE.

It is also useful to have a code metrics tool.

An important metric is the McCabe/Cyclometic metric (recall MSO, else check Wikipedia). If you use Rider you need to install the CyclomaticComplexity plugin.

Visual Studio has a more complete Code Metrics functionality. Along with McCabe it can give many other metrics. The feature is available even in the Community edition, but only the Windows version.

We also allow you to use Github Co-Pilot.

2 Few Important Notes before You Start

Test Flakiness: Random Generator. 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'. 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[4]. Such a generator behaves deterministically when given the same seed. Check the class **Utils.STVControlledRandom** to obtain such a generator.

Test Flakiness: Persistent State. Another source of flakyness is dependency on 'persistent' data, such as a database or a static variable. For example, the aforementioned STVControlledRandom keeps its state in a static variable. When running a set of test methods, keep in mind that they may be executed in a different order when the set is different, or simply because the used unit testing framework makes no commitment on keeping the order the same. This may cause the tests to affect a persistent state in a different order, resulting in flakiness.

[1] https://www.jetbrains.com/rider/

[2] https://www.jetbrains.com/community/education

[3] https://nunit.org/

[4] 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. You can make your random generators deterministic by controlling the seed(s) they use —check the documentation of the class `Random`. When deploying the game for actual users, you can supress the seeds so that they will use random system-seeds.

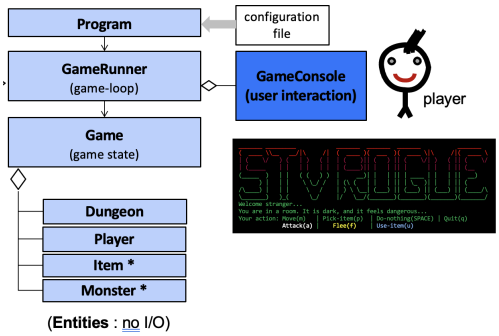

Figure 1. *The architecture in UML-like Class Diagram.*

To avoid this, make sure that you reset relevant persistent states before every run of a test method. See the documentation of the attribute [SetUp] of NUnit :)

Intercepting I/O. For automated system testing, we will use a 'test-agent'. This agent can simulate keyboard commands. The method 'GetUserOrTestAgentInput()' in the class GameRunner controls whether the game reads from your actual keyboard or from the test agent. You can look into this method to see how it works.

LINQ. Check out how to use *Language Integrated Query* in C# to make your code less complicated, e.g. to count the number of healing potions in the player bag:

> (**from** *i* **in** player.Bag **where** *i* is HealingPotion **select** *i*).Count()

Or alternatively, in the functional programming style: player.Bag.Where(i ⇒ i is HealingPotion).Count().

Or, in this case, simply: player.Bag.Count(i ⇒ i is HealingPotion).

3 Architecture

The initial code of STVRogue establishes the simple architecture shown in Figure 1. The class Program serves as the usual main-class from which the game is run. Program will simply call GameRunner; this contains the game main-loop. In this loop, the game shows the game state to the user, and asks the user to decide and enter his/her action. The action is then interpreted to update the game state. This completes a turn, and the loop starts over again.

The state of the game is maintained in the class Game. This method also has the method Game.Update() where you should implement how a single turn updates the game state.

The class GameRunner also holds a pointer to a GameConsole that provides methods for printing texts to the system-console and for reading strings from it. The class Game holds the pointer too, in case you need to do some text-printing from there. Do not use the system-console directly for your console I/O. Use this GameConsole instead.

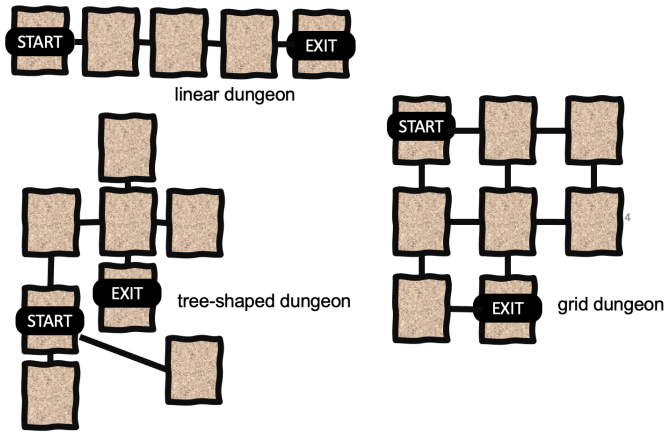

Figure 2. *Some examples of dungeons.*

4 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.

4.1 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 minimalistic class called GameEntity. We will insist that game entities (so, instances of GameEntity) should have *unique IDs*; this will make it easier for you later to debug the game from its UI.

4.2 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 them 'ordinary' room). A dungeon should have one unique start-room, one unique exit-room, and at least one ordinary 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. Self-loop (connecting a room to itself) is not allowed as this tends to confuse users. 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-room*. Figure 2 shows some example of dungeons.

The constructor Dungeon($shape, N, \gamma$) creates a dungeon consisting of $N \geq 3$ rooms that meets certain requirements; see below. It fails (throws an exception) if it cannot construct a dungeon that meets the requirements.

1. A dungeon should satisfy the previously mentioned constraints about its connectivity and the uniqueness of its start and exit-rooms.

2. The parameter *shape* determines the shape of the dungeon. There are three types: *LINEAR*, *TREE*, and *GRID*. A LINEAR dungeon forms a list with the start-room at one of its ends, and the exit-room at the other end. A TREE dungeon contains no cycle, and is not linear-shaped. Furthermore, the exit-room should be a leaf of this tree. A TREE dungeon contains at least five rooms. A GRID dungeon forms a 2D-grid. Figure 2 shows an example. A GRID has at least four rooms.
3. Every room in the dungeon has a *capacity*. It specifies the maximum number of monsters that are allowed to be in the room. Let c be the capacity of a room r
   a. For start and exit-rooms: $c = 0$.
   b. For rooms neigboring to the exit room: $c = \gamma$.
   c. Other rooms have random capacities $c \in [1..\gamma]$.
   Note that the above requirements also imply that $\gamma > 0$ and that the exit-room cannot be a neighbor of the start-room.

The constructor Dungeon is already implemented for you (though not guaranteed to be bug-free).

4.3 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.

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.

4.4 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 (not including the turn when it is used).

Using a potion will consume it.

4.5 Class Game

The class implements the game's main loop, 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 4.2, even impossible to satisfy. The constructor is allowed to fails (it would then throw an exception), if after some k attempts if cannot generate a populated dungeon that satisfies the configuration.

When the dungeon is created, the player should be placed at the start-room of the dungeon. It should be alive, and its HP is equal to HPMax, and >0.

4.5.1 Monster and items seeding. The task to poplate/seed the dungeon with monsters and items is delegated to a helper method called SeedMonstersAndItems (so, your constructor Game($conf$) should call this helper method).

The method SeedMonstersAndItems(M, H, R) randomly populates the rooms in the dungeon with monsters and items.

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 impossible to populate a dungeon with N rooms of max-capacity γ with more than $(N-2)\gamma$ monsters).
The method SeedMonstersAndItems returns true if it manages fullfill the requirements, else it returns false. The requirements are:

1. Every monster in the dungeon should be alive and have HP and AR >0.
2. Every room cannot be populated with more monsters than its capacity allows.
3. Let N_E be the set of neighbor-rooms of the exit-room. Every room in N_E should be populated with at least as many monsters in any non-N_E room. So, for any

[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.

$r \in N_E$ and $r' \notin N_E$, then $|r.\text{monsters}| \geq |r'.\text{monsters}|$ should hold.
4. Let N be the number of rooms in the dungeon. At least $\lfloor N/2 \rfloor$ number of rooms should have no item at all.
5. The start and exit rooms cannot have items (nor monsters, since their capacity is zero).
6. If a room contains healing potion(s), none of its neighbours should contain a healing potion.
7. Rage potions can only be placed in rooms which are "leaves" in the dungeon, and are not the exit-room. (so, how do you recognize if a room is a 'leaf'?) This implies btw that you cannot have a rage potion in a LINEAR dungeon (nor GRID).

4.5.2 Game-update. 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 left to you to decide, as long as everyone gets exacly one action.

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

The main methods of the class Game is update(α). It will advance the game by one turn. This method iterates over all creatures in the dungeon. A monster can choose its action randomly; this will be explained more below. The action of the player is as specified by α.

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 room r. This should be a neighboring room, and furthermore this should not breach r's capacity.
   MOVE is **not** possible when the creature is in combat.

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 4.4.

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.

6. FLEE: the creature flees a combat to a randomly chosen *neighboring* room. Fleeing is subject to a number of conditions listed below.
   a. A monster cannot flee to a room if this would exceed the room's capacity.
   b. The player cannot flee to the exit-room.
   c. In the *Newbie*-mode, the player can always flee.
   d. In the *Normal*-mode, the player cannot flee if in the previous turn it uses a potion.
   e. In the *Elite*-mode, in addition to the restriction in (d) above, the player also cannot flee while it is in the enraged state.

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

5 The Game Loop

The game's main-loop is to be implemented in the class GameRunner, more precisely in the method Game.Run(ϕ). The implementation is not complete :) You should complete it. You can for now ignore the parameter ϕ —this is only relevant for an Optional task later.

This main loop is directly called from the top level class Program, so you can run and try this loop by running Program.

At every iteration of this main loop, the game status is printed to the Console, and then the loop waits for the player's action. The player does the action by pressing a key on the keyboard. The loop should handle invalid inputs given by the user, or if multi-inputs are needed. If the inputs form a valid command, then a command α is constructed and passed to the method Game.update(α) to decide what to do with the command. The loop then advances to the next iteration. This is repeated until the game ends.

When ran, the main loop first 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. Some actions may not always be possible. E.g. using a potion is not possible when the player does not have any. Likewise, fleeing is not always possible. When the player tries to do an action that is actually not possible, your program should not crash. Instead, it should print a message notifying the player that the action is not possible. Importantly, this does not count as his/her action for the turn. The player can retry with another action.

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.

6 The Class Program

The class STVrogue.Program is the main class (the class with the Main method) from where the game will be configured, created, and run. When you start the application, it reads the game configuration from a file (configuration is explained in Section 4.5). It then creates an instance of Game according to this configuration, and run it.

By default the configuration file is:

root`/STVrogue/saved/rogueconfig.txt`

where *root* is the directory where you put the STVrogue git (the directory where you find the `readme.md` of the project).

7 Your Tasks

Your tasks are listed below. All are mandatory, except the optional-variant of Task 7. 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. The method Move(r) (of Monster and Player) and the method Creature.Attack(f) (0.5 pt).
2. The constructor Dungeon($shape, N, \gamma$) (1.5 pt).
3. The constructor Game($configuration$) (1.5 pt).
4. The method Game.Flee(c) (1.5 pt).
5. Finishing the implementation of STV Rogue (2 pt).
6. Test the rest of the game logic (1.2 pt).
7. System-level testing (0.5 - 1.5 pt). Basic form: 0.5 pt. Optionally you can do a stronger system-level testing and its reporting for 1.5 pt.
8. Report (0.3 pt).

Test coverage requirement. For 7.1 - 7.4 and 7.6, all produced tests should deliver 100% code coverage[6] on their test target and all its worker methods (e.g., your tests on Flee(c) should give 100% coverage on this method, and other workers it invokes). For 7.7 we have a separate requirement explained later.

Delegated logic/worker. You may decide to delegate some of the logic of the above listed targets to another class. E.g. in the implementation of *Game.Flee*(c) you might delegate some of the logic to *Player.Flee*(). Keep in mind that this delegated logic/worker should then also be fully covered by your tests.

> 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 gives a rough estimation on the minimum number of test cases you would need to test it (but keep in mind that it won't take delegated logic into accout). The metric gives the number of 'linearly independent' control paths in the method.

7.1 Test Move(r) of Monster and Player. Test Creature.Attack(f) too. (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 to test them (and to fix them if you find bugs). Note that Move(r) of Monster and Player also call Move(r) of the superclass Creature; don't forget that the move of Creature has two branches.

Use NUnit Framework to write your tests.

7.2 Test the constructor Dungeon($shape, N, \gamma$) (1.5 pt)

The constructor is already implemented for you. The intended behavior of this constructor is informally specified in Section 4.2. Formalize its informal specification as an **in-code specification** and then use NUnit **parameterized test** to test the method. Figure 3 shows an example.

The implementation may have some bugs for you to fix.

The class Utils.HelperPredicates contains some help predicates you might find useful. E.g. it contains a predicate to check if a dungeon is linear-shaped. You may also want to play with CoPilot a bit to see if it can generate some assertions for you. You can e.g. first type what you want in a plain language, in a comment, and see what CoPilot then generate out of it. Here is an example of what it produced (in my case):

```
// each room has a capacity between 0 and capacity:
Assert.That(Forall(D.Rooms, r => r.Capacity >= 0 && r.Capacity <= capacity));
```

Note that CoPilot might help saving some typing work, but you cannot blindly trust it. Ultimately, **you** are responsible for the correctness of your solution and that your tests are sensical.

[6] 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. What "statement" means typically depends on the tool you use. For Rider this generally refers to simple statements such as an assignment or a return statement. A whole if-then would count as several statements. E.g. Rider would see if($p||q$) {x++; **return** x} as three 'statements': the guard $p||q$, the assignment x++, and **return** x.

```
[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) ;
          // Note: using AreEqual is here better. Check its doc.
      }
      else {
        // (b) the method should throw this exception when its
        // pre-condition is not satisfied:
        Assert.Throws<DivideByZeroException>(x % y) ;
      }
  }
}
```

Figure 3. *An example of how to write an NUnit test through an in-code specification. Let's imagine we want to test C# remainder operator (%).*

7.3 Implement and test the constructor Game(*conf*) (1.5 pt)

The intended behavior of this constructor is informally specified in Section 4.5. 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.

Your tests should give full coverage on the constructor and its worker methods as well.

7.4 Implement and test the method Game.Flee(*c*) (1.5 pt)

The intended behavior of this method is informally specified in Section 4.5. The logic of this method is not trivial. Implement and test it (1pt).

For 0.5 pt, use NUnit "Theory" to verify your flee-logic on the player in the Elite difficulty. Check that the logic is correct regardless the player HP, the number of monsters in the room, and how long the player has been engared (0..5 turns). Obviously checking this on e.g. all possible values of HP might blow up the number of test cases. So you can instead introduce a domain partitioning.

Check NUnit Documentation: https://docs.nunit.org/articles/nunit/intro.html 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.

7.5 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*) and finish the class GameRunner. The resulting game should be able to run without crashing and has all the asked features. It does not have to be beautiful.

7.6 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). We aim for 90% coverage on the classes under GameLogic. If you have less, give the reason in your report (e.g. unreachable code).

7.7 System Testing (0.5 - 1.5 pt)

System-level testing of computer games is typically hard to automate. STVRogue is no exception. You will have to manually play-test the game to make sure that it works properly.

Having said that, STVRogue has been designed to allow automated testing using a 'test-agent'. You can use this to do automated system-level testing to complement your manual play-testing. To define a test-agent you write a subclass of the class TestAgent and implement the method NextAction(). When used, the game will call this method whenever it would otherwise read the keyboard for the user's input. So, in this method you can program how the agent would drive the game. E.g. it could be to just randomly play the game. See the project Coba_TestAgent to see an example how to hook an agent to the game.

In this task, you will do automated system-level testing using such an agent. You have two options:

- *Basic system-testing* (0.5 pt). Implement a random agent that just randomly interacts with the game (simulating legal and illegal keys). Use this agent to verify that the game is robust (does not crash) on all types of dungeons and on all difficulty modes. Use dungeons with at least 15 rooms. Have your tests to run N times on each game instance, to compensate for the randomness of the agent. You should get at least 70% code coverage on the class GamerRunner.
- *Stronger system-testing* (1.5 pt). Implement a random agent (as above) and a smart agent. I leave it to you to come up with your own algorithm for the smart agent.
  1. (0.5) Use both agents to verify that the game is robust (does not crash) on all types of dungeons and on all difficulty modes. Use dungeons with at least 25 rooms.
  2. (0.5) Also verify that the following sanity properties hold through out the plays: (1) the player's HP never exceeds its HPMax; (2) no creature can remain in the dungeon if its HP has become zero or negative; and (3) the player's KP never decreases. Use the

parameter ϕ in the GameRunner for checking such properties.

3. (0.5) Benchmark the code-coverage performance of your smart agent versus the random agent, over increasing room numbers. You would need to rerun the agents N-times to get an average performance value. Include the result of this benchmarking as a graph in your Report. Also provide a description of how the algorithm of your smart agent works.

7.8 Report (0.3 pt, mandatory)

Make a report containing the items listed below.

1. Team-id and members' names.
2. Mention the location of your tests.
3. Some short instructions how to play the game, so we can check that it works.
4. 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

5. 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($shape, N, y$)		
mcCabe metric (*)	:	...
# test-cases (**)	:	...
coverage	:	...
Game($conf$)		
mcCabe metric	:	...
# test-cases (*)	:	...
coverage	:	...
Dungeon.SeedMonstersAndItems(M, H, R)		
mcCabe metric	:	...
coverage	:	...
Game.Flee(c)		
mcCabe metric	:	...
# test-cases (*)	:	...
coverage	:	...
Game.Update(cmd)		
mcCabe metric	:	...
# test-cases (*)	:	...
coverage	:	...

(*) Also known as the Cyclometic metric.

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

6. Explanation: if your coverage for the targets listed above is below 100%, mention why you failed to get it to 100.

7. If you do the optional "Stronger System Testing" Task, describe the algorithm of your smart test-agent. Present the result of your benchmarking of the smart agent.

8. 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.

8 Submitting

A Blackboard assignment will be created to submit your project.

1. State in the **readme.md** where your unit tests are located. We will run your *Program*-main and all your NUnit tests. Make sure they do not crash.
2. Upload a zip to the Blackboard, containing of the whole project and the pdf of your report. The name of the zip-file should begin with TEAM_N where N is your team-id number. Only one person from each team needs to submit the zip.