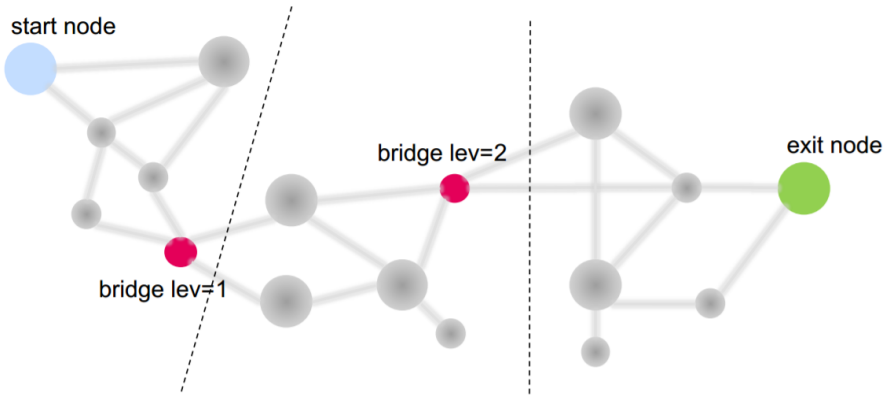
STV Project 2016/17, Iteration 1

Software Testing and Verification

W.van Steenbergen, J.Stolwijk and N.Hendrikx



Informatica

Universiteit Utrecht

Year 2016/2017

Contents

Introduction

Dear Reader,
This report covers our effort and contribution to the first iteration of the main project for Software testing and verification. Also it highlights the results and observations we had. The team members are Woudt van Steenbergen, Jesse Stolwijk and Niels Hendrikx.

Chapter 1

Logical entities in the game

This chapter will cover all the subjects of the mandatory part of this project. We managed to acquire 100 percent block coverage (1.2) over all the mandatory classes. We wrote the unit tests with the platform Nunit for cSharp.

1.1 General

These statistics are covering every file in the project STVRogue. The unit test statistics are including the optional automated tests.

N = total # classes	21
locs = total # lines of codes()	556
locsavg = average # lines of codes()	locs/N = ˜26.5
Mavg = average # methods per class	74/21 = ˜3.5
Mmax = max # methods per class	11
cabe = the total mcCabe complexity	214
cabeavg = average mcCabe complexity per class	6.9

Table 1.1: General statistics

N' = number of classes targeted by your unit-tests	16
T = number of test cases ()	55
T locs = total # lines of codes of your unit-tests	694
T locsavg = average # unit-tests lines of codes per target class	T locs/N' = ˜43.4
E = total time spent on writing and executing tests	64
Eavg = average effort per target class	4
bugs = total number of bugs ever found by testing	218

Table 1.2: Unit Test statistics

1.2 Classes

Particular information for each class.

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (%
Niels_RAGDING 2017-05-18 14_58_23.coverage	56	3.04 %	1785	96.96 %
stvrogue.dll	23	2.43 %	924	97.57 %
STVRogue.Utils	23	9.66 %	215	90.34 %
STVRogue.GameLogic	0	0.00 %	662	100.00 %
Player	0	0.00 %	55	100.00 %
Pack	0	0.00 %	66	100.00 %
Node	0	0.00 %	86	100.00 %
Monster	0	0.00 %	4	100.00 %
Item	0	0.00 %	16	100.00 %
HealingPotion	0	0.00 %	7	100.00 %
GameCreationException	0	0.00 %	2	100.00 %
Game	0	0.00 %	139	100.00 %
Dungeon	0	0.00 %	258	100.00 %
Crystal	0	0.00 %	10	100.00 %
Creature	0	0.00 %	9	100.00 %
Bridge	0	0.00 %	10	100.00 %
STVRogue	0	0.00 %	47	100.00 %
UseItemCommand	0	0.00 %	16	100.00 %
MoveCommand	0	0.00 %	15	100.00 %
Command	0	0.00 %	2	100.00 %
AttackCommand	0	0.00 %	14	100.00 %

Figure 1.1: Block coverage

1.2.1 Dungeon

Dungeon includes the following methods.

- Dungeon() *Constructor*
- genGraph() *Generates random dungeon.*
- shortestPath() *Calculates the shortest path between two nodes.*
- dungeonIsValid() *Checks if the dungeon is a valid dungeon.*
- setNodeLevel() *Calculates the nodes level.*
- MonsterHP() *Sums the HP of all the alive packs.*
- PotionHP() *Sums the HP of all healing potions.*
- disconnect() *Disconnect every node with a lower nodeLevel from the bridge. The bridge is the new start node of the graph.*
- F_Y_shuffle() *Randomly shuffle the list's items.*
- shuffleConnect() *Shuffle a array and connect the nodes.*

- level() *Returns the nodes' nodelevel.*

We achieved 100% block coverage for this class. In the Test class NTest_dungeon are all the test which belong to the dungeon class. The number of unit tests is 13.

1.2.2 Creature

- Monster.Attack() *The monster attack() a creature.*
- Player.Attack() *The player attack() a creature.*

1.2.3 Packs

Multiple Monster form a pack. The move() command can be use to flee.

- Attack() *The pack attacks the player.*
- Move() *The pack to the neighboring node of the pack's current location, after evaluating the capacity.*
- MoveTowards() *Calculates the shortestpath(), move() the pack one step towards the desired node.*

1.2.4 Items

- HealingPotion.use() *Uses the healing potion, the player regenerates HP.*
- Crystal.use() *Uses the crystal, if the player is on a bridge, disconnect the bridge.*

1.2.5 Node

- CountCreatures() *Returns total amount of creatures in all packs on the node.*
- connect() *Connects Node(a) to Node(b), and Node(b) to Node(a)*
- disconnect() *Disconnects Node(a) to Node(b), and Node(b) to Node(a)*
- fight() *Executes the AI fight.*
- getRandomFleeNode() *Get a node where the pack can flee towards.*
- Capacity() *Calculates the capacity of the node.*
- nodeLevel *Returns the nodelevel.*

Bridge

- connectToNodeOfSameZone() *Add the node to the fromNodes and do Node.connect()*
- connectToNodeOfNextZone() *Add the node to the toNodes and do Node.connect()*

1.2.6 Commands

MoveCommand can be use by the player to flee.

- MoveCommand.Execute() *Player move() to a random neighbour.*
- UseItemCommand.Execute() *Player uses a random item.*
- AttackCommand.Execute() *Player attacks a random monster thats on the same node.*

1.2.7 Game

- Update(Command) *Executes the command.*
- SpawnMonsters() *Spawns the monsters after the dungeon is generated.*

Chapter 2

Optionals

This chapter will cover all the optional subjects of this project.

2.1 Testing the combat

An stronger approach of testing than branch or block coverage is Edge-Pair coverage. This method is more strong and it will have an less exploding effect compared to prime-path coverage. You can take each edge pair apart and for each edge pair you can write an specific unit test. Visual studio will only notice the 100% block coverage but you will have to explain for each test which pairs it covers. But to have less test you write a test which will cover multiple edge-pairs. In figure 2.1 there is

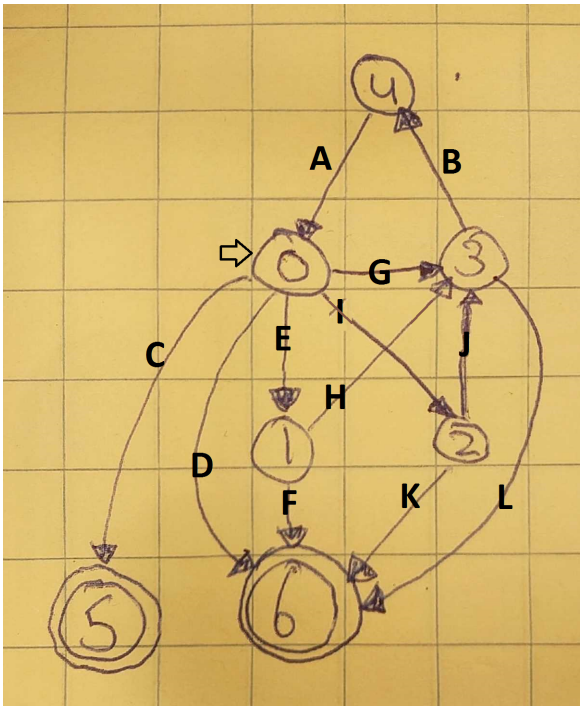


Figure 2.1: simplified Combat CFG

a simplified control flow graph of the combat procedure. Each node number is the same node number as in the CFG inside the project document (Figure 2). Each alphabetic letter is an edge. And we want to cover all feasible edge pairs. Node 5 & 6 are the exit nodes, Node 0 is the start Node. Test T1 : [0, 3, 4, 0, 1, 3, 4, 0, 2, 3, 4, 0] covers the edge pairs [G, B, A, E , H , I, J]. To have the edge pairs C, D, L , F and K is impossible because there nothing after an exit node. We cant cover those for each other. But we can attach these edges to 5 different test versions of T1.
Test suite:

1. [0, 3, 4, 0, 1, 3, 4, 0, 2, 3, 4, 0, 5] = [G, B, A, E , H , I, J, **C**]
2. [0, 3, 4, 0, 1, 3, 4, 0, 2, 3, 6] = [G, B, A, E , H , I, J , **L**]
3. [0, 3, 4, 0, 1, 3, 4, 0, 2, 3, 4, 0, 1, 6] = [G, B, A, E , H , I, J, **F**]
4. [0, 3, 4, 0, 1, 3, 4, 0, 2, 3, 4, 0, 2, 6] = [G, B, A, E , H , I, J, **K**]
5. [0, 3, 4, 0, 1, 3, 4, 0, 2, 3, 4, 0 ,6] = [G, B, A, E , H , I, J, **D**]

These tests will provide maximum pair-wise coverage.

2.2 Automated testing of the dungeon

For automated testing we used FSCheck with the FSCheck.NUnit extension. To create automated tests we need to create a generator that generates a random difficultyLevel for the dungeon. Every test iterates 100 times, for each iteration we create a new Dungeon object (new random graph). Every method in the Dungeon class has an automoted test, see FSCheck_Automated.cs.

2.2.1 Random

We define Random as a process that chooses a value from a collection of values, with the chance of choosing the value = $P(value) = 1/size(collection)$ this is also known as a uniform distribution. To check if our graphs are generated randomly, we need to represent our graph in a Hash. C# has a default hash function for List.GetHashCode(), so we only had to implement the Node.GetHashCode(). see figure 2.7.

Listing 2.1: Generate hash for Node

```
public static int GetHashCode(this Node x)
{
        unchecked
        {
                int hash = 17;
            hash = hash * 31 + x.neighbors.GetHashCode();
            hash = hash * 31 + x.id.GetHashCode();
            hash = hash * 31 + x.nodeLevel.GetHashCode();
            return hash;
        }
}
```

If we want to check if our graph is randomly generated we can call the List.GetHashNode() n times and test if the hashes have a strong corellation with the uniform distribution of seeds used by the pseudo-random generator. We can use the chi-squared test to measure the correlation.

2.2.2 Results

We've found several bugs using automated testing.

- The F_Y_Shuffle() method did not shuffle properly.
- Shortestpath(start,end) failed if start != the dungeon's start node.
- Disconnect() failed because it did no use toNodes from Bridge.

2.3 Mutation test

For mutation testing we used VisualMutator as a plug-in in Visual Studio. We applied mutation to all the classes in gameLogic. All the mutants where killed by the unit tests. In one case the mutant ended in an runtime error. This happens with the mutation #SSDL. In 3 cases a mutant was still alive. In the class Dungeon we had around 50 tests that killed the mutants. This seems rather high, but this is including the optional automated tests. This is a good result! (2.3.1)

Listing 2.2: SSDL error mutation

```
66   -  while (monstersLeft > uint0)
101  +      IL_0263:
102  +          if (local_24.MoveNext())
103  +              goto IL_0223;
104  +      }
105  +      goto IL_03a2;
67  106       {
107  +      IL_0282:
```

2.3.1 Dungeon

For each method in this class I will give an example code of an mutant that was Killed by a test.

Constructor

Example killed modulo mutation.

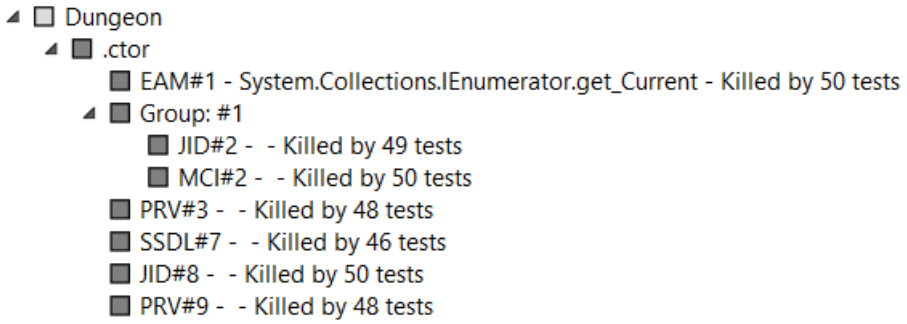


Figure 2.2: Dungeon constructor results

Listing 2.3: Modulo mutation

```
16  - this.zone_size = new int[this.difficultyLevel + 1];
16  + this.zone_size = new int[this.difficultyLevel % 1];
```

genGraph

Example killed multiply mutation.

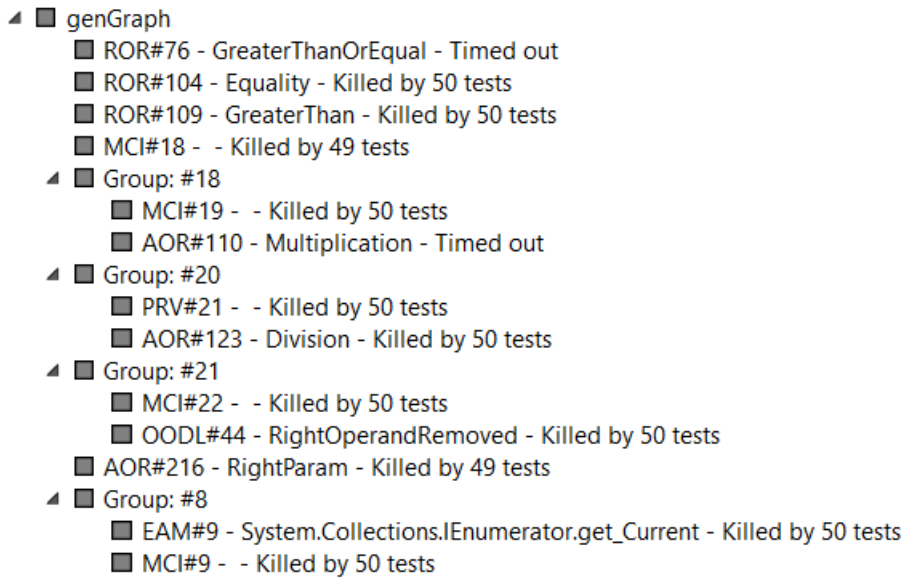


Figure 2.3: Dungeon genGraph results

Listing 2.4: Multiply mutation

```
31 - STVRogue.GameLogic.Bridge b = new STVRogue.GameLogic.
     Bridge(string.Concat((object)(rSize + cnt - 1)));
33 + STVRogue.GameLogic.Bridge b = new STVRogue.GameLogic.
     Bridge(string.Concat((object)(rSize * cnt - 1)));
```

Disconnect

Example killed mutation.

Listing 2.5: Mutation

```
7  -          STVRogue.GameLogic.Node[] allNodes = new STVRogue.
     GameLogic.Node[b.neighbors.Count];
7  +          STVRogue.GameLogic.Node[] allNodes = new STVRogue.
     GameLogic.Node[b.neighbors.Capacity];
```

Level

Example killed mutation.

Listing 2.6: Mutation

```
1    public uint level(STVRogue.GameLogic.Node d)
2    {
   -        return d.nodeLevel;
3  +        return this.startNode.nodeLevel;
4
5    }
6
```

F_Y_shuffle

Example killed mutation.

Listing 2.7: Mutation

```
8   -                if (!(j == i))
10  +                if (!i)
```

These results are a select few of many. In the project zip file we attached a .XML document of all our results.

Chapter 3

Team reflection

In this chapter we cover the roles of every team member and what their main focus was during the first iteration. In general the process of the project went frictionless from the start. For the estimated effort-percentage inspect 3.3.

3.1 Jesse

Implementing/tests for nodeLevel (setlevel) and disconnect in dungeon. Tests for player and implemented the move method. Implemented/tests for command. Added a CountCreatures, capacity and made some small changes to connect and disconnect in node. Movestowards in Pack and all pack tests. Implementing combat(user combat commands, pack decision making, handling exit states) together with Woudt. Finally I created Automated testing tests for all the Dungeon methods and proposed an automated way to test if the dungeon constructor behaves randomly.

3.2 Woudt

Implementing checks for base rules of the game such as the unused potions and monster health points($HP_P \leqslant 0.8 * HP_M$). Spawning of monsters. Spawning of items. Implementing combat(user combat commands, pack decision making, handling exit states) together with Jesse. Testing of said implementations.

3.3 Niels

My main focus from the start was generating the random graph. I Implemented and wrote the following basic entities: Dungeon, Dungeon_is_ok, genGraph, shuffle_connect, F_Y_shuffle, shortestpath. After this I provided 100% block coverage with unit tests for these methods. After this I worked on Mutation Testing, Improved Testing combat, Skeleton report & statistics. I tried to maintain a good overview of progress and planning.

Name	Estimated relative effort-percentage
Jesse	35%
Woudt	30%
Niels	35%

Table 3.1: Effort-percentage