HOGESCHOOL VAN AMSTERDAM

Refactoring source code Sprint 1

Project Game Technology

Jasper Meier & Rosa Corstjens, 500703267 & 500702627

March 2017

Glenn Hoff, Wouter van de Hauw, Michael Fuents Rodriguez, Rosa Corstjens, Guus van Gend, Kevin de Leeuw, Frederick van der Meulen & Jasper Meier

Project Game Technology
Team 5

Under the guidance of
Alexander Mulder, Product Owner
Anders Bouwer, Scrum coach
Eric Klok, Technisch consultant

Refactoring source code Sprint 1

Project Game Technology

Jasper Meier & Rosa Corstjens, 500703267 & 500702627

Game Technology
Hogeschool van Amsterdam
March 2017

1. INTRODUCTION

This report briefly discusses the methods that have been refactored in the first sprint of Project Game Technology. In the upcoming sprints, we plan to set up an automatic quality gate with the program SonarQube. This gate will point out code that needs to be refactored. A standard scenario will be written, which needs to be followed to fix the issues.

In the first sprint we used ReSharper locally to find code to refactor. Issues that we found critical or important have been identified throughout the code and refactored. For each of these issues, screenshots of the code before and after are included. Also an explanation about the taken steps is given. In this explanation we point out what parts of the code we refactored and, if a refactoring method from the site www.refactoring.com is used, which catalog names from this source match these issues.

2. INITIALIZE MEMBER VARIABLES IN CONSTRUCTOR

By initializing all member variables in the constructor, errors can be prevented, since you ensure that the variable has a suitable default value. When no explicit default value exists, '0' can be assigned. The following classes had constructors that are improved by initializing all of their member variables:

- UIManager;
- LevelManager;
- GameManager;
- Character.

There are two different ways in which the default value could be set in the constructor, either in the class's initialization list or in the constructor function itself. The first way is used to refactor the listed classes. An example of how the code before and after:

```
// before
UIManager::UIManager() { }

// after
UIManager::UIManager() : _uiNode(0),
_healthBar(0), _staminaBar(0), _maxWidthBar(0),
_heightBar(0) { }
```

3. DEFAULT CASE HANDLED IN SWITCH

By handling the default case in a switch statement, you make sure the program won't crash when an unsuspected value is encountered. Even though it seems like isn't possible in the current implementation, it may be in a later stage. The only occurrence of a missing default case was in the **BaseNPC** class's **wander** function. The code before and after refactoring is as follows:

```
// before
switch (random)
{
case 1:
        _dirVec.x = 1;
        break;
/* … cases 2-4 left out … */
case 5:
        _dirVec = (0, 0, 0);
        break;
}

// before
switch (random)
{
case 1:
        _dirVec.x = 1;
        break;
/* … cases 2-4 left out … */
case 5:
        _dirVec = (0, 0, 0);
        break;
default:
        _dirVec = (0, 0, 0);
        break;
}
```

4. REMOVING UNUSED INCLUDES

Compiling includes costs processing power and memory, therefore header files shouldn't be included

if they aren't used. The following classes included unused header files:

- Character;
- FloatRange;
- StatRange;
- GameManager;
- LevelManager.

The code before and after refactoring is as follows:

```
// before
#include "Player.h"
#include "BasicEnemy.h"
#include "BaseApplication.h"
#include "CharacterStats.h"
#include "LevelManager.h"
#include <OgreSingleton.h>
#include "UIManager.h"

// after
#include "BaseApplication.h"
#include "LevelManager.h"
#include <OgreSingleton.h>
#include "UIManager.h"
```

5. REMOVING REDUNDANT ARGUMENTLISTS

C-standards state that an empty argument list should be indicated with the void keyword. This is considered redundant in C++ and, since it doesn't add to the readability, it should be avoided. The void keyword indicating empty argument lists only occurred in some methods of the **GameManager**. The code before and after refactoring is as follows:

```
// before
virtual void createScene(void);

// after
virtual void createScene();
```

6. PULL UP CONSTRUCTOR BODY

For this issue a method from www.refactoring.com, namely 'pull up constructor body', has been used. This method implies that code is written in multiple child constructors and can be moved to the constructor of the super class. The class **Character** has a few classes that inherit from it. Most of these classes had their own constructor implementation, which is moved to the **Character** class. The code before and after refactoring is as follows:

```
// before
// super class
Character::Character()
{
    _dirVec = (0, 0, 0);
    movespeed = 50;
    rotationspeed = 0.13;
}

// inherits from Character class
Npc::Npc() { }

// inherits from Character class
BasicEnemy::BasicEnemy()
{
    _dirVec = (0, 0, 0);
    _movespeed = 50;
    _runspeed = 450;
    _rotationspeed = 0.13;
}

// inherits from Character class
Player::Player()
{
    _dirVec = (0, 0, 0);
    _movespeed = 250;
    _runspeed = 450;
    _rotationspeed = 0.13;
    _currentLevel = 1;
    _currentXP = 0;
    _xpTillNextLevel =
CalcXpTillLevel(_currentLevel + 1);
}

// after
// super class
Character::Character(Ogre::SceneNode* pMyNode,
Ogre::Entity* pMyEntity) : _myNode(pMyNode),
_myEntity(pMyEntity), _stats(0), _dirVec(0, 0,
0), _movespeed(100), _runspeed(250),
_rotationspeed(0.13), _isRunning(false),
_currentLevel(0), _currentHealth(0),
_currentStamina(0) { }

// inherits from Character class
BaseNpc::BaseNpc(Ogre::SceneNode* pMyNode,
Ogre::Entity* pMyEntity) : Character(pMyNode,
pMyEntity), _timeSince(0) { }

// inherits from BaseNpc class
BasicEnemy::BasicEnemy(Ogre::SceneNode* pMyNode,
Ogre::Entity* pMyEntity) : BaseNpc(pMyNode,
pMyEntity) { }

// inherits from BaseNpc class
Npc:: Npc(Ogre::SceneNode* pMyNode,
Ogre::Entity* pMyEntity) : BaseNpc(pMyNode,
pMyEntity), _inDialog(false) { }

// inherits from Character class
Player::Player(Ogre::SceneNode* pMyNode,
Ogre::Entity* pMyEntity) : Character(pMyNode,
pMyEntity)
{
    _movespeed = 250;
    _runspeed = 450;
    _currentXP = 0;
    _xpTillNextLevel =
calcXpTillLevel(_currentLevel + 1);
}
```

Note that in the before and after code a new class has been added. This is also a refactoring method from the named source, called 'extract subclass', which will be discussed in the next chapter.

7. EXTRACT SUBCLASS

Again, a method from www.refactoring.com has been used, namely 'extract subclass'. The **Character** class used to implement the function **wander**. Since this function was only used by the **NPC** and **BasicEnemy** class, not by the **Player** class, a subclass had to be implemented. The subclass inherits from **Character**

and will be inherited from by the **NPC** and **BasicEnemy** class. This subclass is called **BaseNPC**, since it will form the base for every friendly and hostile NPC. For now it only implements the **wander** function. The code snippet in the last chapter already showed the constructor of the **BaseNPC** class. Other changes are self-explanatory, such as inherit from **BaseNPC** and implement the **wander** function.

CONCLUSION

The code from the first sprint has been refactored on a few basic issues. Consequently refactoring the code greatly improves the code's readability and consistency. Therefore, a scenario has to be written and automatic quality gates have to be set in place. This way it is ensured that refactoring becomes a continuous process.

For the first sprint, some serious steps have been taken, but – as mentioned – improvements have to be made for the following sprints.