Responsibility Driven Design and Software Modeling with UML

Assignment 1

by

A. Smesseim, S. Baraç, A. el Khalki, A.W.L. Oostmeyer, M. Maton

in partial fulfillment of the requirements for the degree of

Bachelor of Science
in Computer Science

at the Delft University of Technology,

to be presented on 18 September 2015.

Supervisor: Dr. A. Bacchelli

TU Delft Delft University of Technology

Contents

1

THE CORE

1.1.

To properly define the classes we need to produce the CRC-cards, we have to take a look to the requirements of our game. In the this section we will show the necessary requirements and underline the nouns that we are going to use to create the CRC-cards.

1.1.1. NECESSARY REQUIREMENTS

1. The game must show all 60 enemies at the start of the game.
2. The enemies are aligned in rectangle of 5 rows and 12 columns.
3. The rectangle of enemies starts at the top left of the playing field.
4. The player and the enemies must not be able to move outside the boundaries of the playing field.
5. The player must be able to move left and right using the arrow keys.
6. The player must be able to shoot bullets up.
7. If the player's bullet hits an enemy, the enemy disappear from the playing field.
8. The score should be shown above the playing field.
9. There should be three different kinds of enemies, with different appearances. The bottom two rows consist of enemies named "large enemies". The top row should consist of enemies named "small enemies". All other rows should consider consist of enemies named "medium enemies".
10. The player could be able to shoot a bomb up using keyboard input, if the player has any bombs. If a bomb is fired, the number of bombs is reduced by one.
11. If a player bullet hits an enemy bullet, then both bullets could disappear.
12. A UFO could appear at either the top left or the top right of the playing field. The starting point of the UFO is randomly determined, and the time of appearance is also randomly determined. The UFO can only move horizontally. When the UFO appears, will move to the other side of the playing field at the speed of one length unit per second, where the length unit is half the width of the UFO. If a UFOis hit by a player bullet, then a random number of points between 100 and 1000 is awarded to the player. When the UFO reaches the other side of the playing field, it disappears.

1.1.2. EXPLANATION OF THE UNDERLINED NOUNS

The requirements give a good representation of the classes we are going to need to build this game. Furthermore the responsibillities of those classes are in the requirements too. Next we need to extract the classes with their responsibillities.

1.1.3. CLASSES FROM REQUIREMENTS

CLASSES & THEIR RESPONSIBILLITIES

Game The Game-class has the most responsibillities. It must handle the key input using the KeyHandler-class. It must handle the different scenes, e.g. the start-scene, using the Scene-class. It has to create al the enemies and allign them correctly. It must also handle the randomness in the game, e.g. enemy shooting a bullet.

Scene The Scene-class handles the Graphical User Interface of the game. A scene contains a couple of elements, e.g. an enemy, a button etc..

Enemy The Enemy-class must represent an enemy. It has 4 subclasses. It collaborates with the class Intersection for calculating whether a bullet intersects with the enemy.

SmallEnemy The SmallEnemy-class is a subclass of Enemy. The subclasses exist, because of the different points that get added to the players score when the enemies are killed. The different types of enemies also have different sprites.

MediumEnemy The MediumEnemy-class must return a score when killed.

LargeEnemy The LargeEnemy-class must return a score when killed.

UFOEnemy The UFOEnemy-class must return a score when killed.

Player The Player-class must represent a player. It collaborates with Game-class and Intersection-class.

Bullet The Bullet-class must represent a bullet. It collaborates with Intersection-class

KeyHandler The KeyHandler-class must handle the input. It collaborates with Game-class.

PlayingField The PlayingField-class represents the field where everything comes together. It collaborates with the Scene-class and Game-class.

Intersection The Intersection-class must keep track of the intersections between an Enemy & a Bullet and the intersection between a Player & a Bullet.

1.1.4. CRC-CARDS

Game	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Handle user-input Handle layout of scenes Handle playing field Handle randomness	KeyHandler Scene PlayingField

Scene	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Handle Graphical User Interface layout	Game

Enemy	
Subclasses:	SmallEnemy MediumEnemy LargeEnemy UFOEnemy
Superclasses:	None
Responsibilities:	Collaborators:
Represent an enemy	Intersection PlayingField

SmallEnemy	
Subclasses:	None
Superclasses:	Enemy
Responsibilities:	Collaborators:
Represent SmallEnemy Return 40 points if killed	Game Intersection

MediumEnemy	
Subclasses:	None
Superclasses:	Enemy
Responsibilities:	Collaborators:
Represent MediumEnemy Return 20 points if killed	Game Intersection

LargeEnemy	
Subclasses:	None
Superclasses:	Enemy
Responsibilities:	Collaborators:
Represent LargeEnemy Return 10 points if killed	Game Intersection

UFOEnemy	
Subclasses:	None
Superclasses:	Enemy
Responsibilities:	Collaborators:
Represent UFOEnemy Return random points between 100 and 1000 if killed	Game Intersection

Player	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Represent Player	Game Intersection

Bullet	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Represent Bullet	Intersection

KeyHandler	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Handle user input	Game

PlayingField	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Handle the playing grid	Game
Delete enemies when they are killed	Scene

Intersection	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Handle the intersections between Enemy & Bullet and Player & bullet	Game

COMPARISON WITH ACTUAL IMPLEMENTATION

Instead of having a Game class and a Scene class, our implementation has a GUI and a GameScene class. They might seem very similar, but the GUI class does not handle the playing field. This is left to the implemented Engine class, which contains the game logic and the playing field itself. The implemented GUI class also handles the GUI layouts, which is not left to GameScene.

Our implementation does not have a KeyHandler defined, instead it uses EventHandler which is supplied by JavaFX. Intersection is also not a class defined by us, we simple use Rectangles (defined by Swing) to check whether entities intersect.

1.2.

The main classes that are implemented are Engine, GUI, GameScene and Entity. The responsibilities of Engine are creating entities like enemies and bullets, updating the positions of these entities, handling collisions, and adding/deleting entities from the scene. The responsibilities of GUI are starting the window, handling the user input, performing the correct actions based on the user input, and displaying the HUD. GameScene's responsibilities are containing all the Entities and update their positions every cycle, and eventually delete the entities. Engine calls methods of GameScene (to add or remove entities from the scene, and to update their positions), and Engine creates Bullets, Enemies, a Ship. GUI call methods of Engine (to pass on inputs and receive outputs), GraphicsContext (used to draw elements to the window), Stage (used to control the window). GameScene calls methods of DrawableEntity (to draw the elements), and Entity (to initialize an added Entity and add the EntityDestroyedListener).

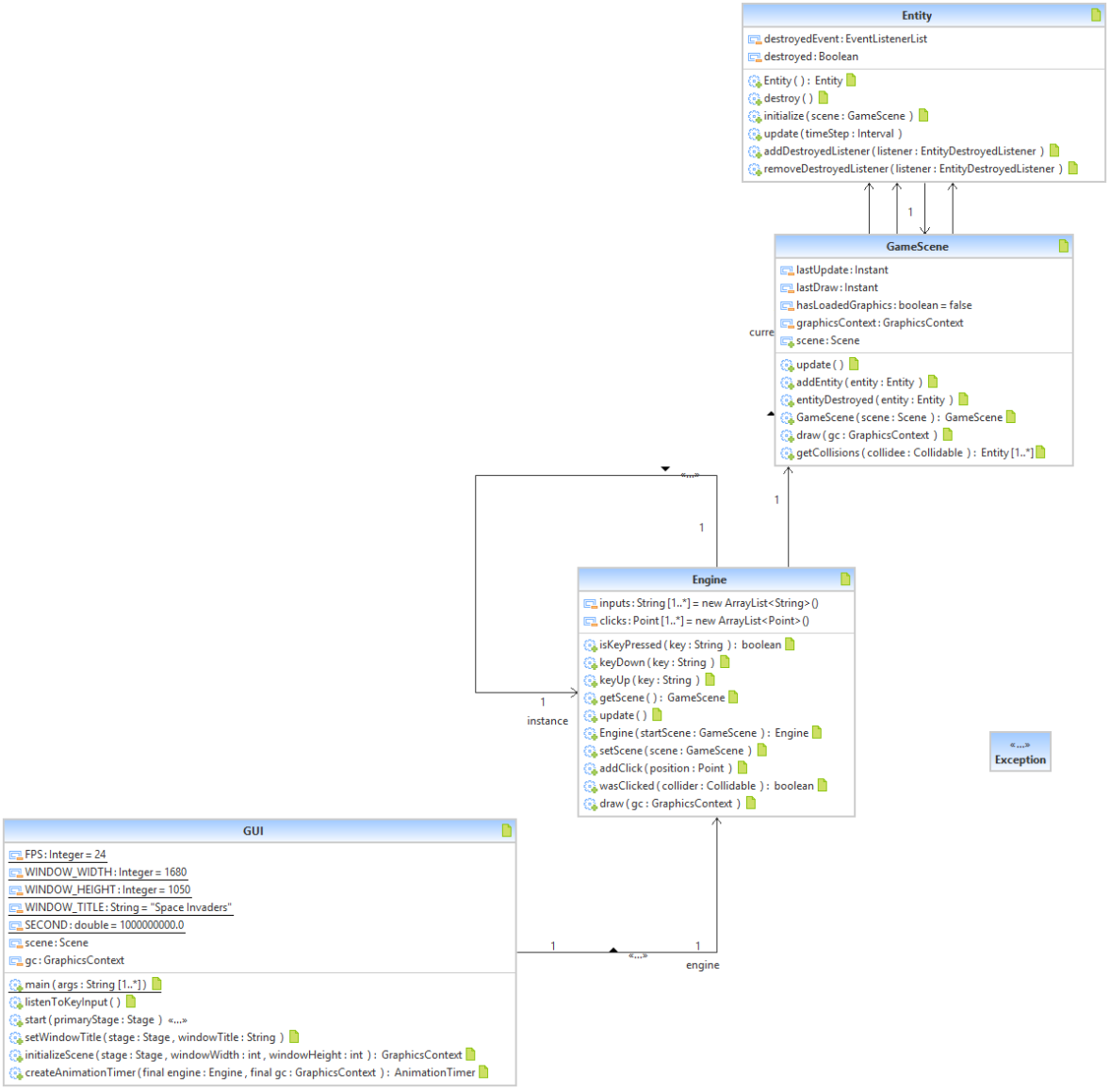
1.3.

Other classes have less responsibilities than the classes mentioned above. This is not the fault of the other classes, but rather the fault of the main classes. The main classes try to do a lot of things, which is not good design. It might be better to refactor functionalities to other classes. To give a concrete example, the Engine decides when an Enemy should fire bullets, and the Engine checks whether a Bullet has hit another Entity.

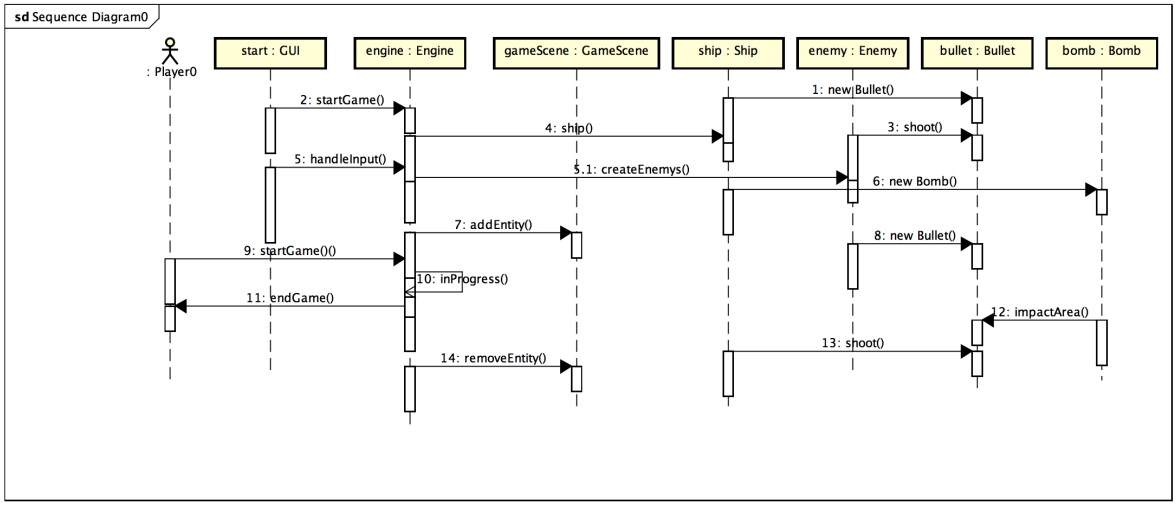
This makes the `update()` method of Engine ridiculously long and hard to maintain. An enemy can perfectly fine decide for itself when to fire bullets, so why not move the responsibility to the enemy? This makes it easier to maintain, easier to understand.

In short, our goal now is not to remove/merge classes with little responsibilities, but rather move some responsibilities from the main classes to other classes.

1.4.



1.5.


sd Sequence Diagram0
: Player0
start : GUI
engine : Engine
gameScene : GameScene
ship : Ship
enemy : Enemy
bullet : Bullet
bomb : Bomb
1: newBullet()
2: startGame()
3: shoot()
4: ship()
5: handleInput()
5.1: createEnemys()
6: new Bomb()
7: addEntity()
8: new Bullet()
9: startGame()()
10: inProgress()
11: endGame()
12: impactArea()
13: shoot()
14: removeEntity()

2

UML IN PRACTICE

2.1.

Aggregation means that the subclass can exist independently of the super class. With composition this is not the case. So the subclass cannot exist without the super class. Examples of aggregation in our project are:

- SpriteEntity—Actor,Bullet: The SpriteEntity class creates the sprite onto the object. Actor and Bullet are subclasses of SpriteEntity. But when we remove SpriteEntity it is still possible for the Actor and Bullet to exist but they won't have sprites.

Examples of composition in our project are:

- Enemy—SmallEnemy,LargeEnemy,UFO: The Enemy class contains the attributes which the all the enemies have in common. So when you remove the Enemy class it is not possible for any enemy to exist.
- Projectile—Bullet: The Projectile contains the shared attributes of all Projectiles. So when we remove the Projectile class a Bullet cannot exist because a Bullet cannot exist if a Projectile doesn't exist.

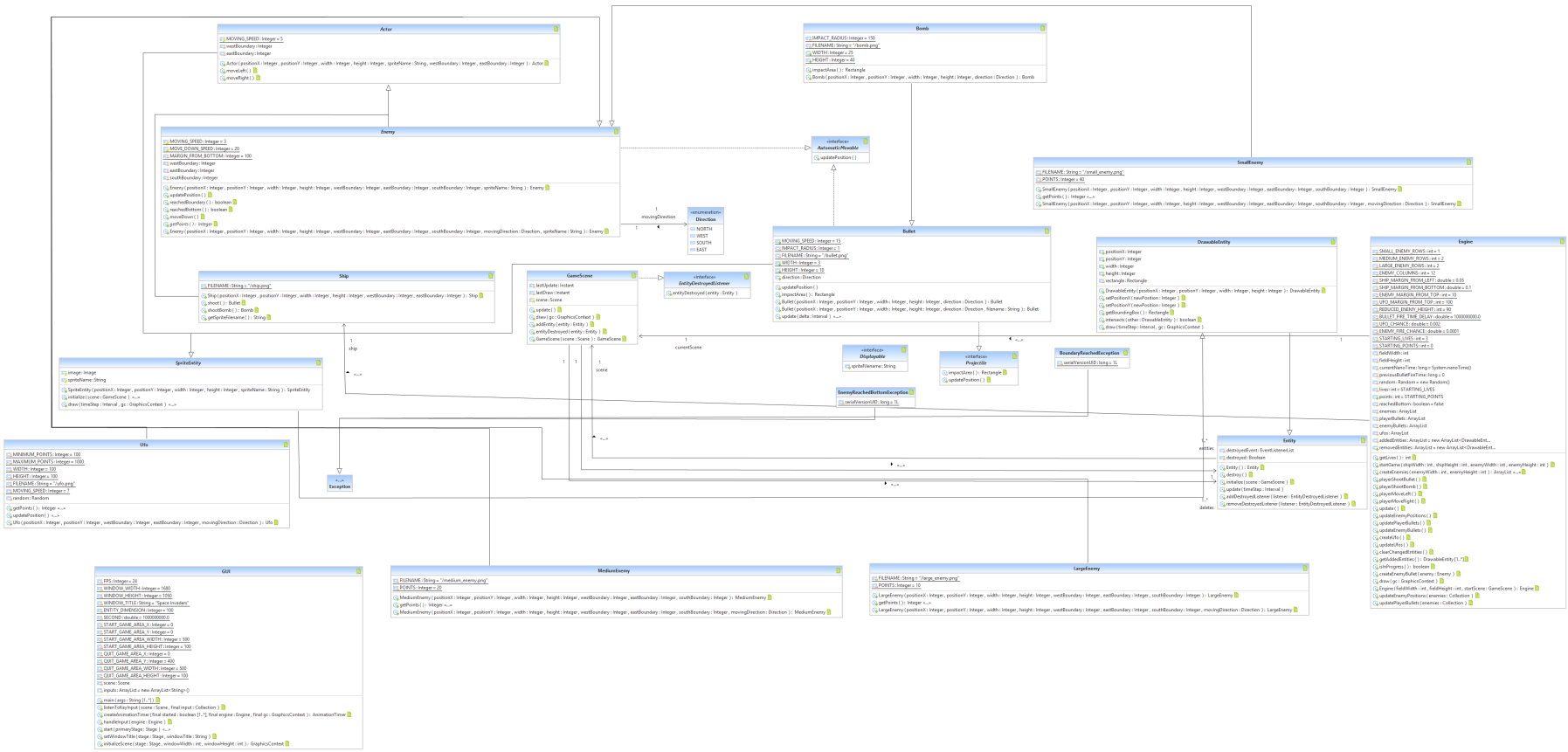
2.2.

We don't use any parametrized class in our source code. A parametrized class comes in handy when a class makes use of objects, but the functionality does not depend of the type of object. Parametrized classes are often used in data structures, for example ArrayList<T>. The ArrayList does not care about the type of T, so it makes sense to make that class generic, so other types could be used too for insertion into the ArrayList. Parametrized classes should be used in a UML diagram when the project use parametrized classes. It is important to specify this in the UML diagram so other developers know the generics exist and can make use of them.

2.3.

The UML diagram of all hierarchies in the source code is shown in figure 2.1. SmallEnemy, MediumEnemy, LargeEnemy and Ufo are all subclasses of Enemy, because other classes usually do not care what the type of the Enemy is. Bomb is a subclass of Bullet. They are both Projectiles, because they behave similarly.

Figure 2.1: The UML diagram of all hierarchies of the source code



3

SIMPLE LOGGING

3.1.

3.1.1. FUNCTIONAL REQUIREMENTS

For the logging feature of the game Space Invaders, the requirements regarding functionality and service are grouped under the Functional Requirements. Within these functional requirements, four categories can be identified using the MoSCoW[1] model for prioritizing requirements. Because the logger is a relatively small implementation, only the 'Must Haves' and 'Should haves' categories are used.

MUST HAVES

1. The logger must output the actions of every game entity. Each individual description of an action is called a log.
2. The format of a log for moving actions is: `<timestamp> - INFO: <type> moved from (<oldX>, <oldY>) to (<newX>, <newY>)`, where `<timestamp>` is the time the action was performed (conforming ISO 8601), `<type>` is the type of entity that moved. `<oldX>`, `<oldY>`, `<newX>`, `<newY>` are the old x-coordinate, the old y-coordinate, the current x-coordinate and the current y-coordinate, respectively.
3. The format of a log when a bullet is fired is: `<timestamp> - INFO: <type> fired a <bullettype> at (<X>, <Y>) in the direction <direction>`, where `<timestamp>` is the time the action was performed (conforming ISO 8601), `<type>` is the type of entity that fired the bullet, `<bullettype>` is the type of bullet that is fired (either a bullet or a bomb). `<X>`, `<Y>` and `<direction>` are the x-coordinate and y-coordinate the bullet was fired from, and the direction the bullet is fired to, respectively.
4. The format of a log when a bullet has hit an entity is: `<timestamp> - INFO: <type> is hit by a <bullettype> at (<X>, <Y>)`, where `<timestamp>` is the time the action was performed (conforming ISO 8601), `<type>` is the type of entity that was bit by the bullet, `<bullettype>` is the type of bullet (either a bullet or a bomb). `<X>`, `<Y>` and `<direction>` are the x-coordinate and y-coordinate the bullet was fired from, and the direction the bullet is fired to, respectively.
5. The preferred output stream (where the description of the actions will be written to) must be specifiable by the caller. In the implementation of the logging feature in the game, the logger will print all logs/descriptions to stdout.

SHOULD HAVES

6. A log level of a log/description should be specifiable by the caller. The log level is the priority of the particular log/description. At least three levels should be specifiably, namely Info, Warning, and Error. In the implementation of the logging feature in the game, all logs of actions will have the log level Info.
7. The minimum log level that warrants writing the log/description to the output stream should be specifiable. In the implementation of the logging feature in the game, the logs that have the log level Info, Warning or Error will be written to the output stream.

3.1.2. NON-FUNCTIONAL REQUIREMENTS

In addition to the functional requirements specified in chapter 3.1.1, the logging feature should also adhere to several non-functional requirements, outlined in this chapter. These requirements will not affect the functionality of the logger, but will ensure that the logger is gradable by the instructors of the course TI2206. The non-functional requirements of the logger are:

8. The logger must be implemented in Java.
9. Responsibility driven design must be used to design the logger.
10. A UML corresponding to the logger must be created.
11. A source code of the logger, in addition of the corresponding UML diagram and analysis of the requirements must be delivered on September 18, 2015
12. Scrum methodology must be applied.
13. The source code must be hosted on GitHub as a public repository.
14. The version control used for developing this game must be Git.
15. The tests must be automated using JUnit.
16. This project must use Maven for build automation.
17. This project must use Travis CI for continuous integration.
18. This project must employ the following static analysis tools: Checkstyle, PMD and FindBugs.

3.2.

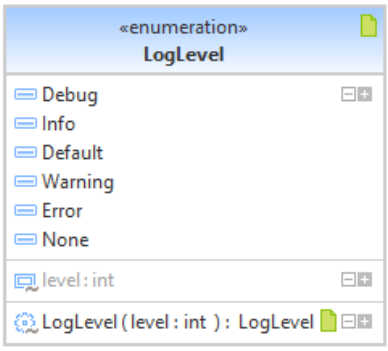
3.2.1. CRC CARDS

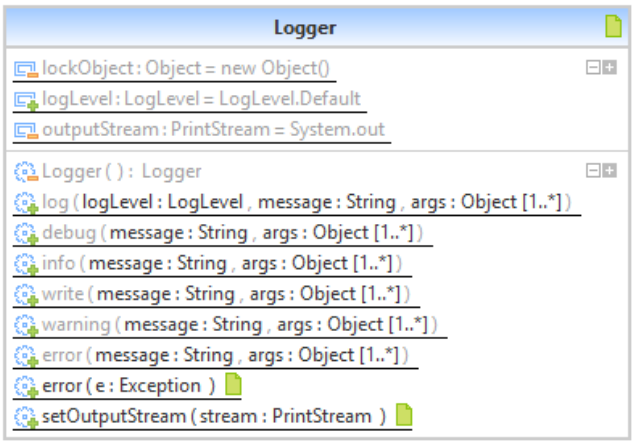
Logger	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Writing logs (strings) to the output stream.	LogLevel

LogLevel	
Subclasses:	None
Superclasses:	None
Responsibilities:	Collaborators:
Represent the priority of a particular log.	

Note that a Log itself is not a class, since it is functionally no different that a standard String.

3.2.2. UML DIAGRAM





BIBLIOGRAPHY

[1] S. Ash, *Moscow prioritisation briefing paper,* DSDM Consortium (2007).