Assignment 1

Project Runner [wip]

Group 5

TI2206 Software Engineering Methods
of the Computer Science curriculum
at the Delft University of Technology.

L. van Aanholt levi.vaan@gmail.com 4084101
M. Wolting matthijswolting@gmail.com 4356063
M.E. Kuipers marcelkuipers1996@gmail.com 4389042
N.E. Hullegien nilshullegien@gmail.com 4389069
H.M. Yeh H.M.Yeh@student.tudelft.nl 4386027

Contents

1 Exercise 1

1.1 CRC cards

The first requirements talk about a main menu screen from which you can start and exit the game, so we chose “Main Menu” as our first CRC card. From the main menu you can do two things: Start the game and exit the game, these are its two responsibilities. When you start the game you will go to a new screen, which we called the “Game screen”. This is the only collaborator of the “Main menu”.

Since we have already defined two types of screens it is useful to already define a subsclass, namely “Screen”. This class does not yet have any responsibilities or collaborators.

The next few requirements talk about what a player is able to do in game. In this case “Player” refers to an entity within the game (the character that is controlled by the user). This entity is the subject of the CRC card called “Player”. The character is able to jump, it will get pulled down by the gravity logic of the game, it should not fall through the environment and it should die when it touches an obstacle or falls into a void. These are its responsibilities. The player interacts with two other objects, namely the Environment (which it cannot fall through) and the Obstacle (which should kill the player when he touches it). These are its collaborators.

Now we define the “Environment” card. Moving from right to left is its sole responsibility. The environment has no collaborators, since it does not interact with other objects, it is only interacted with.

We also define a card for an “Obstacle”. The sole responsibility is to always have the same location relative to the environment. To do this the obstacle will need to interact with the “Environment” object, which will be its collaborator.

In order for the obstacles to be placed in the game we will need an object which does that. This object comes in the form of the “Obstacle spawner”. Spawning obstacles is its responsibility, which also makes the obstacle object its collaborator.

We now have defined three types of different objects that can spawn in the world, namely “Player”, “Obstacle” and “Environment”. Because of this we define a superclass called “World Object” which has all three as a subclass. As of now the “World Object” does not have any responsibilities or collaborators.

The last few requirements deal with what happens when the player dies. We create a new screen call “Game Over Screen”. This screen’s responsibilities are to pop up when the player dies and to allow the user to start the game over again or to go to the main menu.

Main Menu	
Superclasses: Screen	
Subclasses: -	
Responsibilities	**Collaborators**
Start actual game Exit the game	Game Screen Game over screen

Player	
Superclasses: World Object	
Subclasses: -	
Responsibilities	**Collaborators**
Perform jump action Get pulled down by gravity Don't fall through the environment Die when below a certain height Die when in contact with an obstacle	Environment Obstacle

Environment	
Superclasses: World Object	
Subclasses: -	
Responsibilities	**Collaborators**
Move to the left	-

Obstacle	
Superclasses: World Object	
Subclasses: Obstacle spawner	
Responsibilities	**Collaborators**
Move along with the environment	Environment

Game Over Screen	
Superclasses: Screen	
Subclasses: -	
Responsibilities	**Collaborators**
Pop up when game is over Go to main menu	Main Menu

Game Screen	
Superclasses: Screen	
Subclasses: -	
Responsibilities	**Collaborators**
When the player died, it should show the game over screen. Spawn and show the environment Spawn the obstacles	Game over screen Player Environment Obstacle spawner

Screen	
Superclasses: -	
Subclasses: Game screen, Game over screen, main menu	
Responsibilities	**Collaborators**
-	-

World object	
Superclasses: -	
Subclasses: Obstacle, Player, Environment	
Responsibilities	**Collaborators**
-	-

Obstacle Spawner	
Superclasses: -	
Subclasses: Obstacle	
Responsibilities	**Collaborators**
Spawn obstacles in environment	Obstacle Environment

For the biggest part this is also how we have designed our project. Some differences: In reality we used multiple “FloorTile” objects to make up an environment instead of using one “Environment” object. These “FloorTile” objects need to be spawned just the same as “Obstacle” object. Because of this we didn’t use an “Obstacle Spawner”, but instead a “Spawner”, which spawns both obstacles and floorTiles. We also used a container for all “World Object” objects, called the “World”. Because of practical reasons we also created an “CollisionChecker” object, so that collisions between objects only need to be defined in one place.

1.2 Main classes

CollisionChecker	
Superclasses: -	
Subclasses: -	
Responsibilities	**Collaborators**
Check if a player collides with an object in the world Check if a player has fell in the void (out of the world)	Player World

MainGameScreen	
Superclasses: Screen	
Subclasses: -	
Responsibilities	**Collaborators**
Starts the game Quits the game	World

WorldObject	
Superclasses: -	
Subclasses: -	
Responsibilities	**Collaborators**
Create various obstacles	-

Spawner	
Superclasses: -	
Subclasses: -	
Responsibilities	**Collaborators**
Spawn objects created by the world object class in the world	WorldObject World

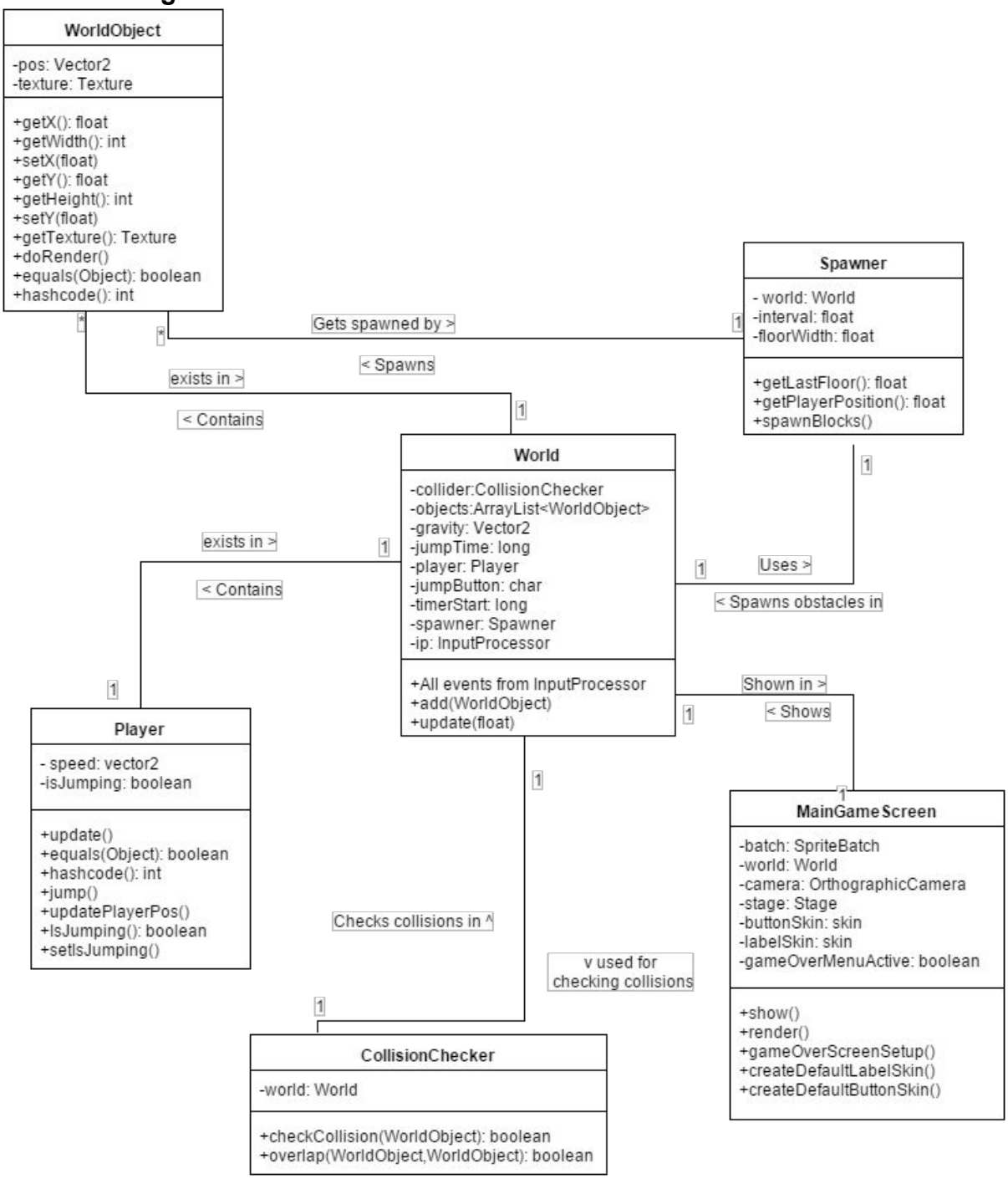
World	
Superclasses: -	
Subclasses: -	
Responsibilities	**Collaborators**
Update the movement of the environment and the objects	WorldObject

Player	
Superclasses: -	
Subclasses: -	
Responsibilities	**Collaborators**
Jump when necessary	-

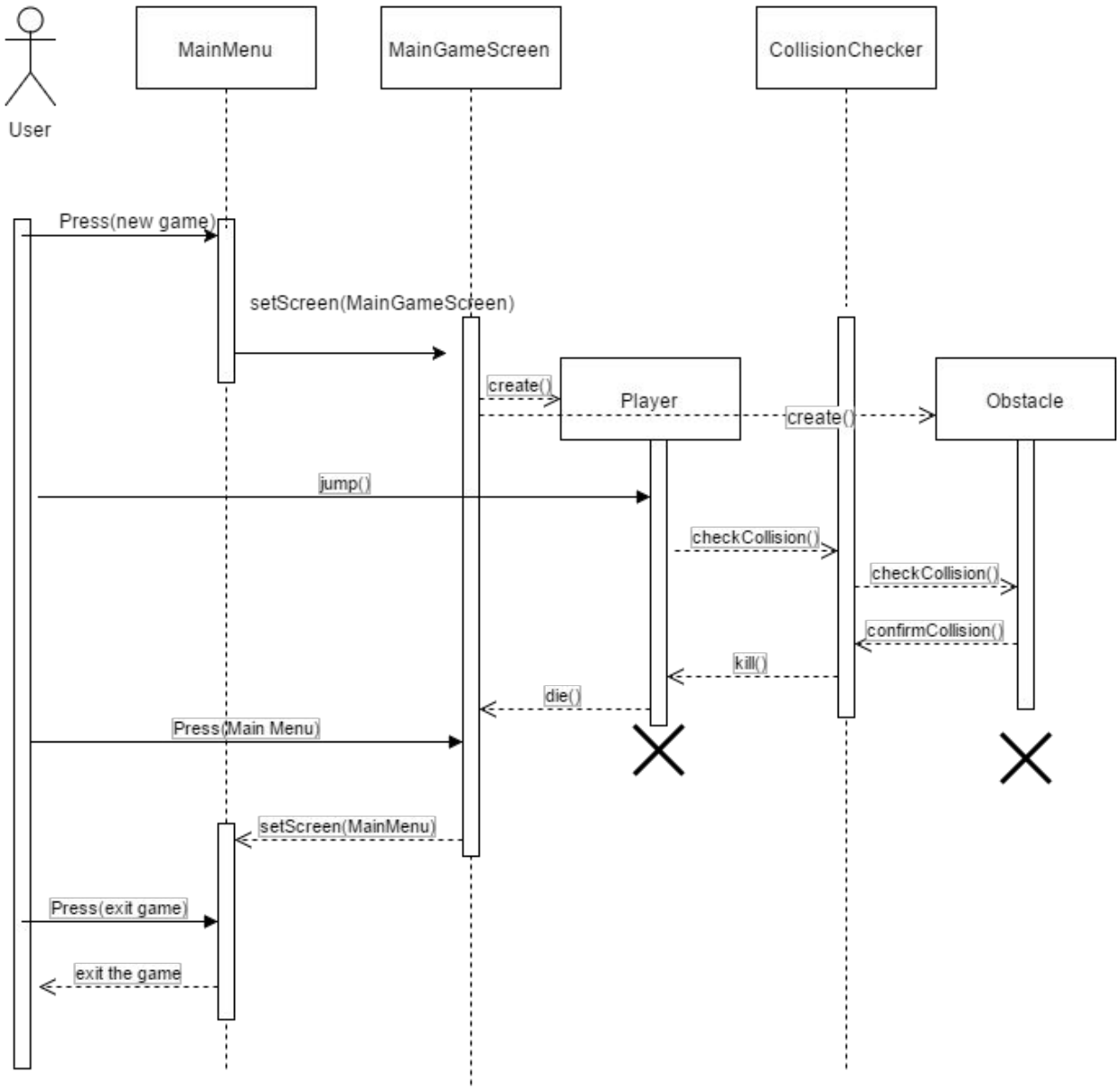
1.3 Other classes

Some classes are not as important as the main classes. The “FloorTile” and “Obstacle” classes are examples. This is because these classes don’t really have a lot of responsibilities for themselves and are only acted upon. We however should not change these classes, because although they don’t have a lot of responsibilities they are still an essential part of the game. All other classes we have implemented are important as well, even though they are also not specified as main classes (the assignment forces this distinction, which cannot really be made). Our project does not really have a huge number of classes and each class has distinct responsibilities and collaborators.

1.4 Class diagram



1.5 Sequence diagram



2 Exercise 2

2.1 Composition and Aggregation

Composition is a relationship between two classes. One of the two classes A is the owner of the other class B. When class A doesn't exist, class B can't exist as well.
Aggregation is a relationship between two classes. One of the two classes A is the owner of the other class B. These two classes are independent of each other. So as a result of that they class B can exist even if class A doesn't exist.

The classes in our project that use aggregation are the World, WorldObject, EndlesRunner and TextureCache classes. The World contains various WorldObjects that belong in the World, but these two classes exist independently.
The EndlesRunner contains one TextureCache that belongs to the EndlesRunner. It's here also the case that these two classes exist independently
The classes in our project that use composition are World, CollisionChecker and Spawner classes. The CollisionChecker and Spawner classes are part of the World class. They can't exist independently, because without the World the CollisionChecker can't check the collisions of the World and the Spawner can't spawn objects into the World.

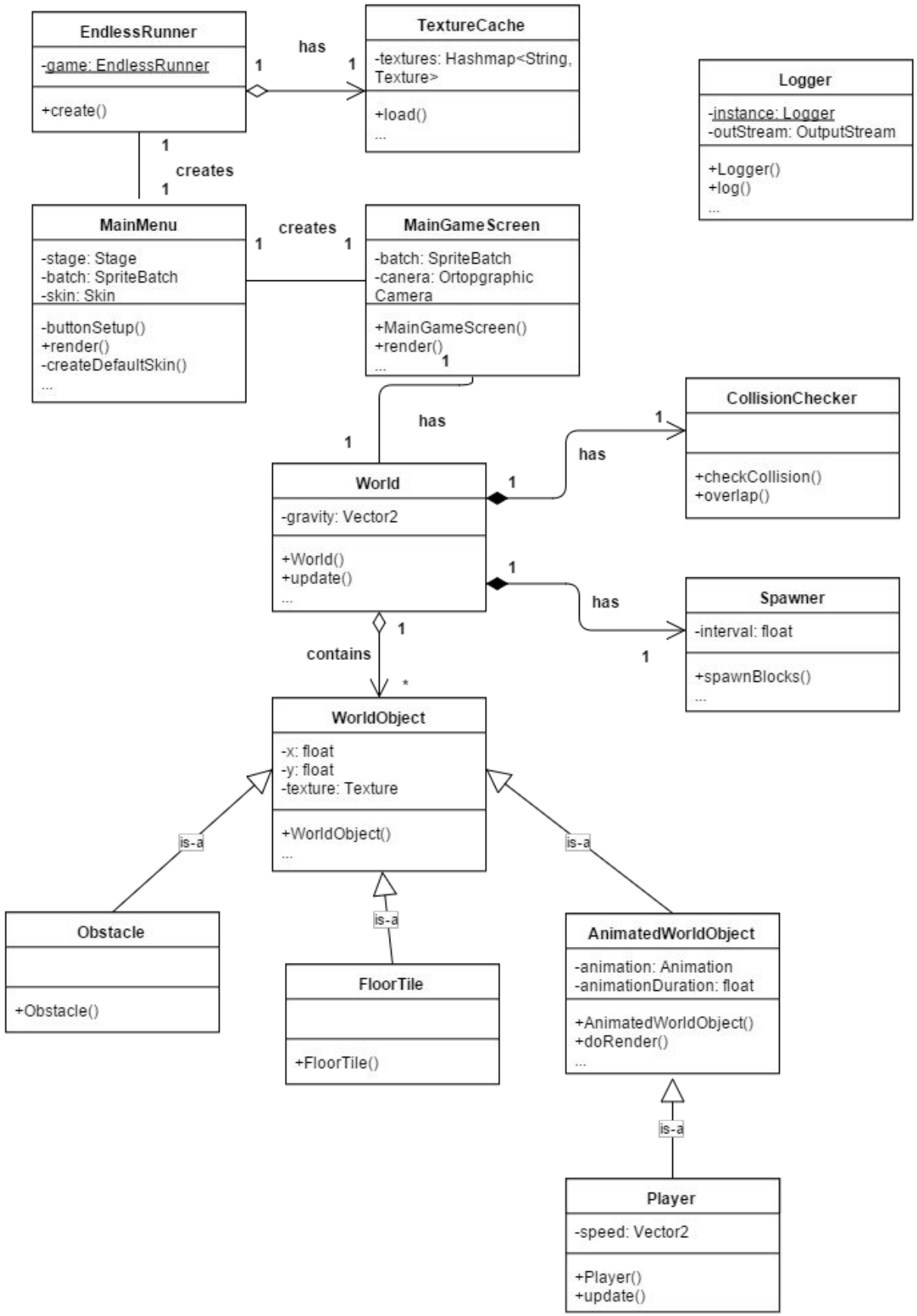
2.2 Parameterized classes

There are no parameterized classes in our source code.
Parameterized classes are used when your classes make use of inheritance and you need to be able to store the superclass or subclasses in a new certain storage class, like a LinkedList. This way you don't need to keep track of every object what kind of class it is, but the parameterized class will take care of this for you. In these cases it is better to make use of a parameterized class.

2.3 UML

The only class hierarchy that exists out of more than one class is the WorlObject class hierarchy. This was done in order to be able to make different WorldObjects like the FloorTile, Obstacle and the AnimatedWorldObject classes. This way all these objects could be easily added to the World in an ArrayList of WorldObjects. Also the Player was an AnimatedWorldObject in order to make it easier for objects in the World that would ahve an animation to add that to the World.
No hierarchies should be removed, because these hierarchies are important for the game to be able to have and add objects.

UML Class Diagram



3 Exercise 3

3.1 Requirements

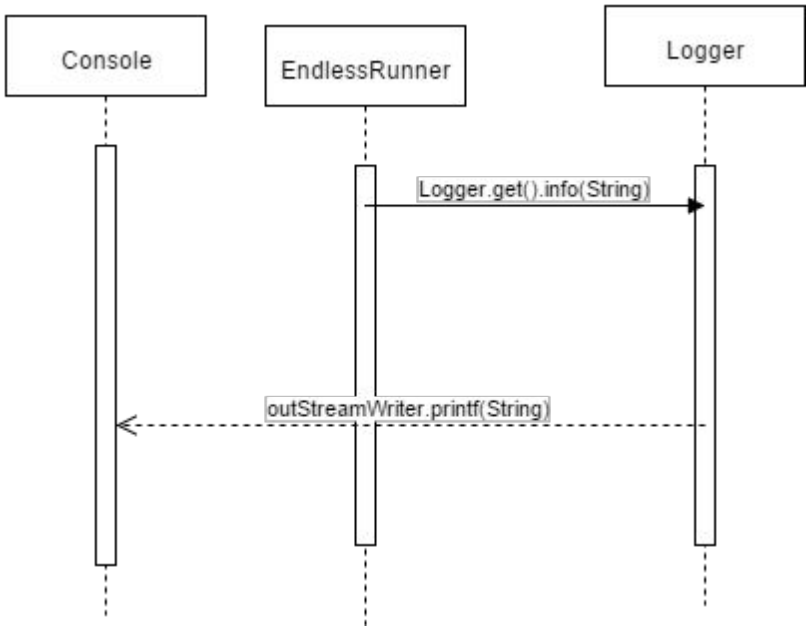
- The logging system must log messages that are passed to it
- The logging system must be a dedicated instantiable class with a singleton instance for general use
- The logging system must have multiple log levels: debug, info, warning, error
- The logging system must have a configurable logging threshold, which is read from a property file, but can be overridden in code
- The logging system must be able to log to either the console or a file
- The logging system must accept a tag for each logging call to make filtering out all messages related to a subsystem easier
- The logging system must output log level, timestamp, tag and message

3.2 Responsibility Driven Design

CRC Card

Logger	
Superclasses: -	
Subclasses: -	
Responsibilities	**Collaborators**
Log output to console or file	-

Sequence diagram



UML