Department of Computer Science
Umeå Universitet

January 22, 2019
Utgåva: 1.0

5DV135 HT18
Application delevopment in java
Documentation AntiTD

Role	**Name**	**Email**
Teacher	Johan Eliasson	`johane@cs.umu.se`
Mentor	Anders Broberg	`bopspe@cs.umu.se`
Mentor	Elias Åström	`eam@cs.umu.se`
Mentor	Jakob Lindqvist	`jakobl@cs.umu.se`
Mentor	Klas af Geijerstam Unger	`klasa@cs.umu.se`
Mentor	William Viktorsson	`williamv@cs.umu.se`

Javadoc: `https://bit.ly/2T15aeV`
Git-repo: `https://bit.ly/2EFHlXa`
Tidsplan: `https://bit.ly/2rIEMeq`

Jonas Sjödin `id16jsn@cs.umu.se`
Marcus Jonsson `c17mjn@cs.umu.se`
Martin Hedberg `c17mhg@cs.umu.se`
Petter Skogh `id16psh@cs.umu.se`

Contents

List of Figures

1 Introduction

This paper describes a project where a game has been created based on given specifications by a customer. A description of the final game will be described as well as technical solutions made in the different areas of the game.

2 User Manual

2.1 Game Description

The game is about sending troops through a level from start to goal. To stop the troops from getting to the goal towers are semi-randomly placed beside the road that tries to kill the troops when they are in range. Getting troops to the goal will reduce the enemy's health and the goal with the game is then to get the enemy's health to zero in as small amount of time as possible. To achieve the goal, the user has to strategically send out different types of troops that have different stats and characteristics. To help the troops some maps have tiles with effects on them such as boost and heal.

2.2 Manual

2.2.1 Compilation

To compile this game you will need to have Java 8 or later installed on your machine. Navigate to a folder where you have all the source files in a subfolder called src. You will also need an empty subfolder called bin. This subfolder will be used to store the compiled files before they are compressed into a jar file so it is fine to delete this subfolder after the jar process is done. To compile the code run the following command in your terminal which is in the same directory as the src and bin folders.

```
$ javac -d bin -sourcepath src src/antitd/AntiTD.java
```

After this you run this command to put all the compiled files into a jar file.

```
$ jar cfm AntiTD.jar src/Manifest.txt -C bin .  -C src levels.xml
-C src graphics/images/ -C src validate.xsd
```

Now you can delete the src and bin subfolders.

2.2.2 Running the game

When you have compiled the game into a jar file you can run it by simply type the following command in the terminal:

```
$ java -jar AntiTD.jar [levelsfile]
```

NOTE: To connect to the highscore database its important to have the attached mysql driver called "mysql-connector-java-5.1.47.jar" in the same directory as the compiled jar.
Levelsfile is a optional parameter to run the game with custom maps that is stored in a xml.
Some system also allows you to run the jar by double click the jar file.

2.2.3 Game

When starting the game the user is greeted by a map selection screen `(Figure 1)`. Choosing a map will start a new game on that map `(Figure 2)`. When a new game is started towers will start spawning with an even time interval.

Each available troop is presented as a button with its cost, health and speed next to it in the lower panel of the game. To spawn a troop the user clicks on its button.

The game information is presented to the left in the lower panel with credits, enemy health and the time that has passed.

The current start position has a yellow dashed circle and the available start positions have a S on them.

The upper panel contains:

- a `Menu` button with:
  - ”`New Game`” : Shows the level selection screen, allowing a new level to be started.
  - ”`Restart`” : Restarts the current level.
  - ”`Pause/Resume`” : Will pause or unpause the game if it is unpaused or paused respectively.
  - ”`Quit`” : Closes the game.
- an `About` button that displays information about who the creators are and when the game was created
- a `Help` button that displays information about how the game is played

The troop types are:

- Fast
  - A fast troop with low cost and low health
- Tank
  - A slow troop with medium cost and high health
- Banker
  - A troop with medium health and medium cost that generates credits as long as it’s alive
- Teleporter
  - A high cost troop that is immortal and can place a teleport on the road. The teleport is placed when the user clicks the button for the teleporter spawn again after spawning the troop.

After a game has been completed, a window containing the current level's scorelist will be shown, as well as the alternative of saving the game session's score to it under a given name.

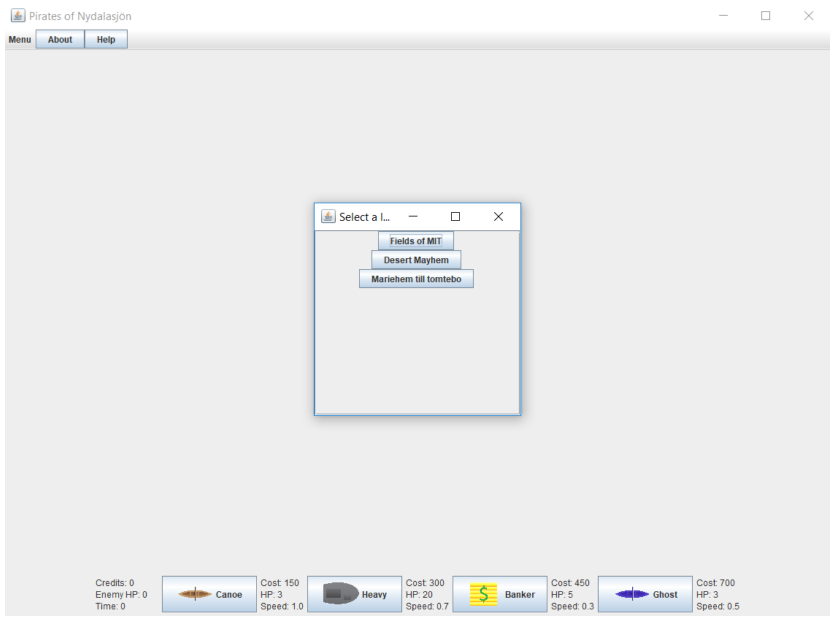

Figure 1 – Level select

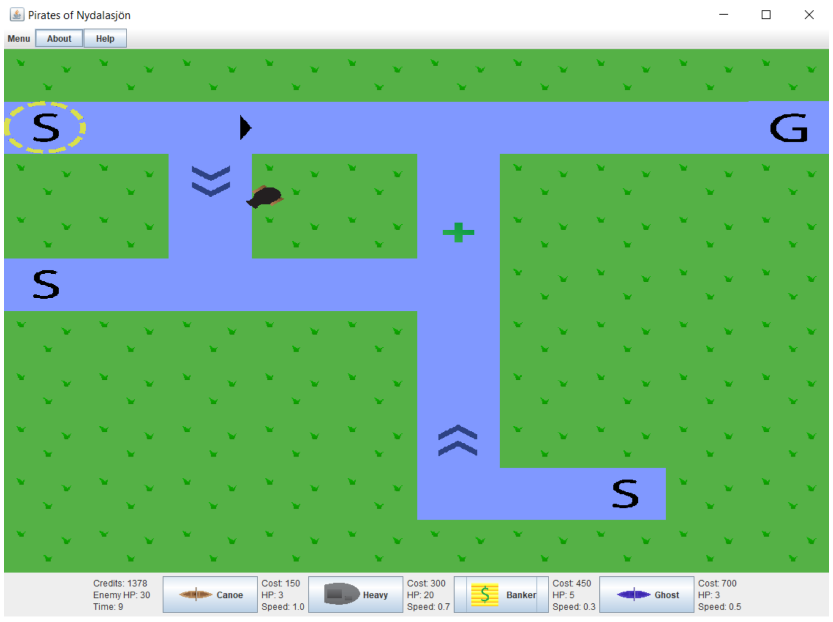

Figure 2 – The game

2.2.4 Map

To create a Map file it has to use a specific syntax to be able to be read by the map parser. Every map file is then read by the validator which validates the `XML` with a `XSD` file. If the given `xml` file is not correct it prints to stderr an error message and throws an exception forcing the program to quit.

1. The root tag must be called **<maps></maps>**
2. Each game map should be put in individual **<gameMap></gameMap>** tags. They should have the attributes **name** which is the name of the map, **width** which is the width of the map **height** which is the height of the map and **ground-type** which is the standard ground type of the map.
3. Each game map is created by using a number of preset tiles, to create a tile it should be created inside a **<tile/>** tag with the only required attributes **x** and **y** of the squares integer coordinates in the map. It also has a some optional attributes, **road-direction** which tells the game that this is a road and it can have the value "e", "s", "w", "n" which is the direction of the road. If the road has multiple directions it can be entered by splitting the text with a '|', like following: road-direction="e|s|w". It also may have the attribute **special** which can have multiple of the following variables start, goal, speed, health and teleport by using the '|' character to split them. The last optional attribute is ground-type which sets the ground type of the tile. It can be one of the following: grass, desert.

```
<maps>
    <gameMap name="Fields of MIT" width="3" height="3" ground-type="grass">
        <tile x="0" y="0" road-direction="e" special="start"/>
        <tile x="1" y="0" road-direction="e" special="health"/>
        <tile x="2" y="0" road-direction="e" special="goal"/>
    </gameMap>
    <gameMap name="Map nr 2" width="3" height="3" ground-type="desert">
        <tile x="0" y="0" road-direction="e" special="start"/>
        <tile x="1" y="0" road-direction="e|s" special="health"/>
        <tile x="2" y="0" road-direction="e" special="goal"/>
        <tile x="1" y="1" road-direction="s" special="speed"/>
        <tile x="1" y="2" road-direction="s" special="goal"/>
    </gameMap>
</maps>
```

Figure 3 – An example .xml file containing two 3x3 sized maps

För att en XML fil ska kunna användas måste den valideras mot följande XSD schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
    <xs:element name="maps">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="gameMap" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="tile" maxOccurs="unbounded">
                                <xs:complexType>
                                    <xs:attribute name="x" type="xs:int"/>
                                    <xs:attribute name="y" type="xs:int"/>
                                    <xs:attribute name="road-direction"
                                                  type="xs:string"/>
                                    <xs:attribute name="special"
                                                  type="xs:string"/>
                                    <xs:attribute name="ground-type"
                                                  type="xs:string"/>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                        <xs:attribute name="name" type="xs:string"/>
                        <xs:attribute name="width" type="xs:int"/>
                        <xs:attribute name="height" type="xs:int"/>
                        <xs:attribute name="ground-type" type="xs:string"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

3 System description

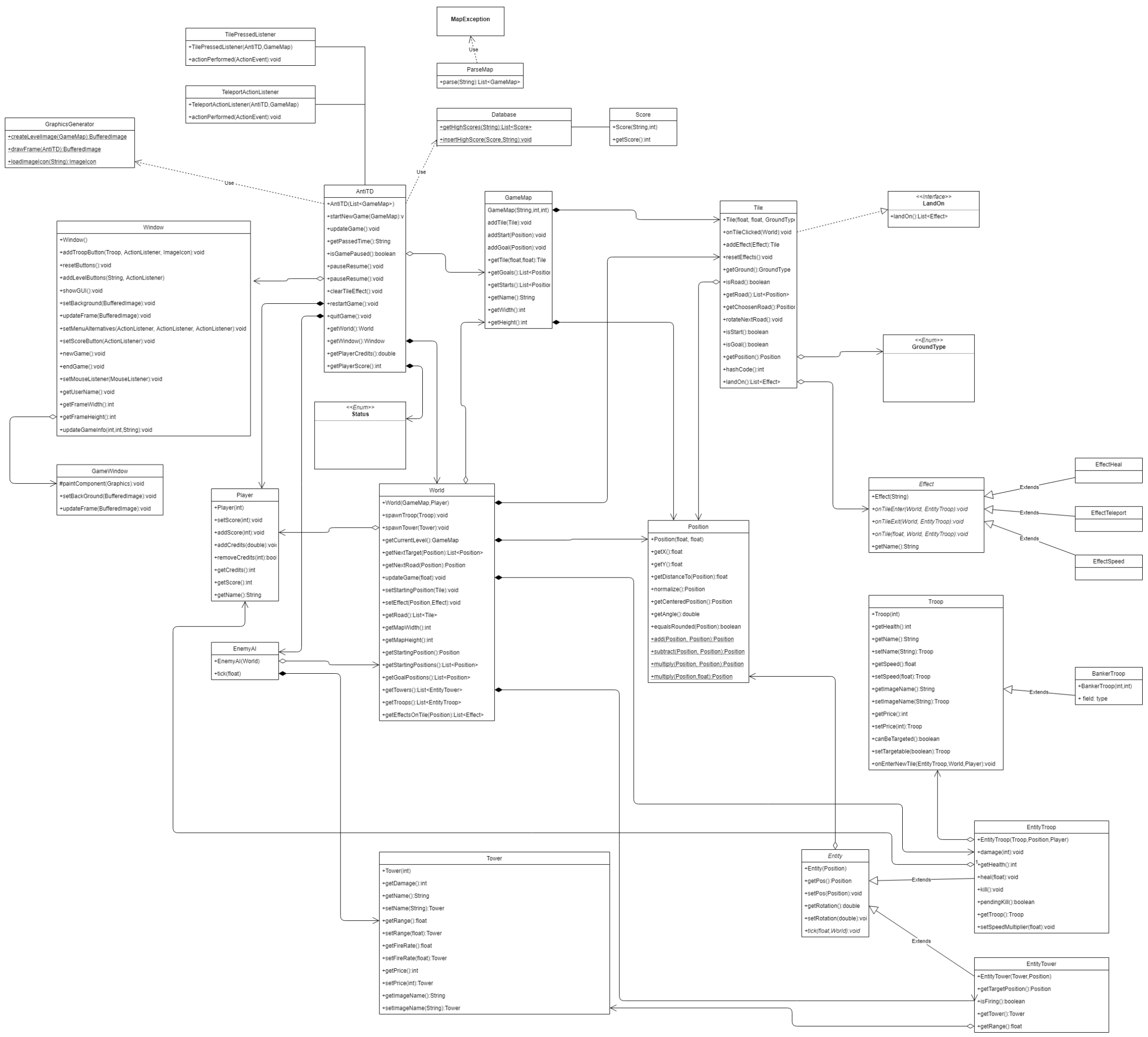

Figure 4 – UML-diagram of the system

Our implementation of AntiTD runs by the class AntiTD:s main method which creates the game and runs it. On creation it sets a few action listeners to the GUI:s graphical interface which makes the buttons work. It also initializes the game. When new game is created in the GUI a thread is started on the `startNewGame()` method. This first quits the last game and initializes a new game and makes it update. In each update it first checks if the game is paused and if it is it waits for it to unpause. Then it checks if it should quit and quits if it should, this happens if the player wants to restart or start a new game. Then it updates the game, the opponent and the GUI. It then checks if the user has won the game. If it has it breaks from the loop, else it continues. Then it represents the user with a High Score list if it can connect to the database.

The `updateGame()` method is one of the most interesting methods in our implementation and it works as following. It checks all its troops and if it is a pending kill. If it is a banker it kills it and removes 1 from the int telling how many active bankers there are. If it is another troop it just kills it. Else it ticks the troop and updates its position etc. Then it ticks the towers making them shoot at targets and then it adds time execution relative amount of credits to the player.

4 Packages

4.1 antitd

4.1.1 AntiTD

The AntiTD class is the main controller class. It handles the whole game and is responsible for updating everything accordingly. When the AntiTD is created it will take an array of maps to play. It will then create a window for the user to operate in. When the window is done it will wait for the user to choose a map, AntiTD will then create a World using that map and begin playing the game. It will create a thread for the graphics update and a thread for the logic in the game. It also has methods that will handle things like pausing, quiting and restarting.

4.1.2 eventHandler

EventHandlers have been added to menu alternatives and spawn buttons for troops on the user interface and functions as following:

- Menu Alternatives:
  - startNewGame(GameMap level): ActionListener for ”New game” alternative in menu. Listener will essentially restart the game with new level from parameter, which is the pressed button’s corresponding level in the level selection screen.
  - restartGame(): ActionListener for ”Restart” alternative in menu. Listener will quit current game and start a new one on the currently selected level.
  - pauseResume(): ActionListener for ”Pause/Resume” alternative in menu. Listener will pause or unpause depending on a status value which is set to the instance’s active status every time the eventHandler is used.
  - quitGame(): ActionListener for ”Quit” alternative in menu. Listener will do a system.exit with code 0.
- Troops:
  - spawnTroop(Troop newTroop): ActionListener for troop spawn buttons on the lower interface panel. Listener will spawn a troop corresponding to the pressed button.
- Other eventHandlers:

- TeleportActionListener(AntiTD atd): ActionListener for special teleport troop, spawning a teleport troop if there is none on the field, or transforming the already existing troop to a teleport if there are.
- TilePressedListener(AntiTD gameInstance): MouseListener for game view. It gets a scale from 0-1 corresponding where the user clicked the screen relative to the window size. It then calculates which tile were clicked and checks if it is a startPosition or a t-section/crossroad. If the clicked tile is the former it sets the game state's start position to the clicked tile. If it is the latter, the direction in the t-section/crossroad will change, which is indicated by an arrow.

4.2 database

This package contains two classes, `Database` and `Score`.

4.2.1 Score

`Score` sets a username and a score value on instantiation and afterwards they can only be gotten and not set.

4.2.2 Database

Database is a static class with a private constructor meaning it cannot be instantiated and should therefore only be used statically. It has two constant variables which should be set to the correct database configuration. It has two public methods, `getHighScores()` which takes a map name as an argument and retrieves the corresponding high score list for that map. If it cannot retrieve the highscores from the database an exception is thrown which is expected to be caught and handled. The other method is called `insertHighScore()` and takes a score and a mapname as an argument. The score is then inserted into the database with the mapname. If the score could not be inserted it throws an exception which is expected to be caught and handled.

4.3 entities

This package contains all the classes responsible for having troops and towers in the game.

4.3.1 Entity

Entity is a abstract class that represents a entity that lives in the world. The entity class stores a position and a rotation that is a common feature along all entities in the world. The Entity class also has a abstract method `tick` which takes two arguments, a float that represent the time difference since the last tick and the world in which the entity resides.

4.3.2 EntityTroop

EntityTroop is an implemented version of Entity that represents a troop in the world. The EntityTroop itself is responsible for the logic of the troops in the world. The logic is run from the implemented tick method from Entity. In `tick`, the entity will first control if it is in a goal, hence has completed the map. If so, the entity will add score to the player and kill itself. Otherwise the entity will start

to update all of its states. It will begin with running any effects that is currently effecting the entity, which the entity gets from the tile it's currently on. Then it will validate its position to see if it's has moved or is on a invalid tile. Then it moves on to update its velocity and then apply the velocity to its world transform.
The EntityTroop also contains the troops current health, a reference to which kind of troop the entity is and if the entity is alive. These states are set from public methods that are used both from within the EntityTroop and from other classes.

4.3.3 EntinyTower

EntityTower is a implemented version of Entity that represents a tower in the world. The tower entity handles the tower logic, same as EntityTroop. It does this by implementing the `tick` method from Entity. Every tick the tower entity will search for a target in range and update its firing state. The firing state is used to decide when the towers shooting effect is rendered. Because the shooting will occur over multiple frames, the entity need to hold the same state for a period of time. Here we have chosen to have it render its shooting laser for 0.3 seconds, when that time is over it will call `damage` on the targeted EntityTroop and then return a non shooting state.

4.3.4 Tower

The Tower class is the base class for every tower. It holds no logic and is purely a data storage class that is used to define new tower. To create a new tower, all you have to do is create a public static tower in the tower class and then only use the reference to that static tower instance when you want to use it. The tower class holds data such as damage, price, range, rate of fire, name and image file name.

4.4 entities.troops

4.4.1 Troop

Troop is a data storage class similar to the Tower class. However it can also contain logic that is called upon when the troop moves to a new tile. This is added to be able to have troops that have different behaviour. Troop also contains a troop name, troop image file name, movement speed, price and a boolean if the troop can be targeted. The troop class also works like Tower in the sense that you create a public static troop. This makes it so that you only need to define the troop here and the game will automatically add a buy button for it.

4.4.2 BankerTroop

The banker troop is a simple implementation of a troop with its own behaviour. It overrides the `onEnterNewTile` method that is called once a entity enters a new tile, and in it adds a line of code that adds credits to the player.

4.4.3 Effect

Effect is an abstract class that is used to implement effects that will effect the troops on the playing field. It hold one attribute that is the effects name and takes that as a parameter in the constructor. There are three abstract methods, `onEnterTile` which is called when the troop enters the affected tile, `onExitTile` which is called when the troop exits the tile and `onTile` which is called consecutive when the troop is on the tile.

4.5 entities.effects

4.5.1 EffectHeal

EffectHeal is an implemented effect that will heal the troop by 1hp every second while the troop is on the tile.

4.5.2 EffectSpeed

EffectSpeed is an implemented effect that will set the troops speed to double when it enters the tile and set it back to the default when it exits the tile.

4.5.3 EffectTeleport

EffectTeleport is an implemented effect that will teleport the troop when it is on the tile. It will teleport the troop 5 tiles forward in the map. If there is a crossroad in those steps it will move in the direction that the crossroad is set to at the time of teleport.

4.6 gui

The view of the program consisting of two classes, `Window`, which is responsible for the player interface and `GameWindow`, the game screen showing the user what happens in the game.

4.6.1 Window

Window is the main view that the user will see when the game starts, it is divided into a `menuBar` containing a menu and user guidance, a reserved space where `GameWindow` will be shown and a `JPanel` holding game information and JButtons used to spawn new troops in game, as well as information about each available troop. The functions of all buttons and menu alternatives are set by public setters inside `Window`. In the menu, following alternatives are available for the player:

- New game: Opens up a `JFrame` containing all available levels as `JButtons`, each with a unique `ActionListener` set by using `addLevelButton(String name, ActionListener listener)`.
- Restart: An `ActionListener` set to this alternative will run in `AntiTD`, restarting the game with `restart()`.
- Pause/Resume: An `ActionListener` set to this alternative will run in `AntiTD`, pausing or resuming the game with `pauseResume()`.
- Quit: An `ActionListener` set to this alternative will run in `AntiTD`, shutting down the programme with `quitGame()`.

After a game has ended, a public method `endGame(ArrayList<Score> scoreList)` can be called in `Window`, which shows a new window congratulating the player, showing the current score list for the current map taken in as a parameter, and gives the player the alternative to enter a name and store the score of the game session.

4.6.2 GameWindow

GameWindow is a class extending JComponent used to store the game's graphics using two `BufferedImage` objects. One of the objects is used to store the games background, which is set using a public setter that sets the image to the method's `BufferedImage` argument using `repaint()`. The other `BufferedImage` object is used to store all active images, i.e. any troops, towers and other activities occurring in game, and is updated using `repaint()` the same way the background is set.

4.7 graphics

The graphics package contains the directory `images` and the class `GraphicsGenerator`.

4.7.1 images

The game's graphics are stored in the images map.

4.7.2 GraphicsGenerator

The graphics generator is responsible for generating all game graphics. The class has one method for creating the background/level image and one for creating the image with entities and special states of tiles.

The picture for the level is created by going through the GameMap for the level. For every tile a picture is drawn based on the tile's ground type, for example grass or desert. If the tile is a road the corresponding picture for the road will be drawn on the tile or if it is a start the start picture will be drawn.

The method for drawing entities and special states will draw a start marker on the worlds current start position, arrows for the current direction of crossroads, effects of tiles as well as troops and towers with their current position and rotation.

4.8 map

The map package contains five public classes handling read of `*.xml` maps and getting the attributes of those maps later on via getter methods.

4.8.1 GameMap

A single `GameMap` which takes a name and its size upon creation. Tiles, goals and starts can be added through package-private methods. All variables can be accessed through getter methods. When a tile is requested, the user sends the tile that should be accessed's position as an argument to `getTile()` and if there is a tile on that position it retrieves it and returns it otherwise null.

4.8.2 LandOn

`LandOn` is an interface which is implemented by `Tile`. It has only one method, landOn() which returns an ArrayList of effects that this tile contains.

4.8.3 Maps

A static class with a private constructor meaning it should only be accessed statically containing one public method, `parse()`. It takes the path to the `*.xml` map as an argument and parses that file creating all the containing gamemaps with its specifications set. Each map has a set size which is used to make sure that all tiles in that map is initialised by creating standard tiles on all tiles that are not set. All the attributes of the map and tile are read and set. After creation of each map it checks for simple errors, like no goal, no start, no name and invalid size.

4.8.4 Tile

An individual tile in a GameMap containing information about that specific tile. This information is its position, its ground type, if it is a road and if so where it leads to, if it is a start or a goal and if the tile has any set effects like speed boost, heal etc.

4.9 player

This package contains the two players in the game, the user and the AI.

4.9.1 EnemyAI

This class has one public method tick which is a round of play for the enemy AI. If it has enough credits it creates a random tower and places it within range to the road in the game. It also increments its credits.

4.9.2 Player

This class has two variables, score and credits. Score is the score of the user i.e. how many troops that have reached the goal and credits is the amount of credits of the user which is used to purchase troops. Purchasing troops can be done by using the method `removeCredits()` and giving the amount of credits that should be removed as an argument. If the requested credits to be removed are greater than the amount of credits that the player has it returns false and no credits are removed.

4.10 primitives

The primitives package contains the class `Position` and the two enums, `GroundType` and `Status`

4.10.1 Position

The position class has a x and a y. The class is used to give objects positions and can then be compared using methods such as `getDistanceTo` or `equals`.

4.10.2 GroundType

Contains the values `grass`, `desert` and `water`.

4.10.3 Status

Contains the values `ACTIVE`, `QUIT` and `PAUSE`.

4.11 world

The world package contains the class `World`.

4.11.1 World

The World class is used to get and set information about the the game state. The class contains what map is being played, the troops and towers that exist in the world, the start and goal positions of the map as well as the road tiles and the player.

Spawn methods exist for troops and towers respectively that gives given troop or tower a position in world and adds them to list of troops or towers. The spawn method for troops sets troop to current start position. The spawn method for towers finds a new position for the tower that isn't on a road or another tower.

The method for updating the world will either kill/remove troop or "tick" troops which updates the troop itself, the method will also "tick" towers and update credits.

5 About the code

5.1 Design pattern

The program is structured by the MVC model. The class `AntiTD` acts as the controller by mainly communicating between `World` and `Window` and updating the logic of the game.

5.2 Thread Safety

This program uses one EDT thread drawing the created graphics, a main thread initialising everything about the game and another thread which runs the programs updater method `updateGame()`. This thread is created when each new game is started or restarted. The old game thread is killed when a new game is started. Since the EDT doesn't know anything about the game itself there should not be any race conditions and our program should therefore be correctly threaded.

5.3 Limitations

5.3.1 Map limitations

The maps biggest limitation is that there is no control on the maps to see if they are winnable or not. There is only a check to make sure that the map contains at least one goal and one starting point, but no check that makes sure that those to are connected with a road. There is also no control to see if the road leads to a dead end, which will kill the troops as they have nowhere to go.
The maps will also have to be stored in a xml file that is named exactly "Map.xml" and lays in the root of the games jar-file.

5.3.2 Gameplay limitations

Because how the movement of the entities are implemented, there is a speed limit on how fast a entity can move. If the entity moves too fast when approaching a corner, it could overshoot this corner and miss the road. The same is true if the frame-rate is very low. When this happen the troop will simply die, for simplicity's sake.

6 Code Testing

During the development all classes were tested to make sure that they did what we planned them to do. The tests were a good guidance on what our module should do and to make sure that they did what they were supposed to do. Since the plan of our program changed during the project our structure of the program changed and therefore some unplanned classes appeared and some disappeared making us write some tests after the classes were created. Some classes like `enemyAI` were hard to test since they just had void functions which were public. They were tested by running the game and in other test.

7 GUI Testing

To ensure that the program functions as per specifications, various tests have been made on the stability and usability of the game. Following are the tests made to the program:

7.1 Test 1 - Startup and closing of game

The game acts as expected when first starting up the game and then shutting it down using the quit alternative in the menu, any open windows (level select, help, about etc.) closes as well.

7.2 Test 2 - Restarting while changing level

Selecting one level first and then changing it does not affect which level is restarted when using the restart alternative in the menu. After changing to a new level five times and restarting the game, the currently selected level is always restarted.

7.3 Test 3 - Spawning troops while the game is paused

Pressing the spawn button of any available troop while the game has been paused by using the pause alternative in the menu does not spawn any troops, nor does it reduce the current credit count.

7.4 Test 4 - Playing while not being able to connect to database

Playing the game offline, or without connection to database, results in the victory screen's score list informing the user that the game could not connect to the database.

7.5 Test 5 - Playing on several game instances of the simultaneously

Having several instances of the game run and playing them simultaneously does not in any way affect the individual game instances running.