Tower Defense

Generated by Doxygen 1.9.8

Chapter 1

Source content

This folder should contain only hpp/cpp files of your implementation. You can also place hpp files in a separate directory `include`.

You can create a summary of files here. It might be useful to describe file relations, and brief summary of their content.

Chapter 2

Test files

It is a common practice to do unit tests of each class before you integrate it into the project to validate its operation. In this folder, you can create your own unit test files to validate the operation of your components.

> It might be a good idea to also take some notes about the tests since you are required to report these in the final report.

2.1 Unit Tests

2.1.1 Test of MyClass

Involved Classes:

Test File:

Results:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

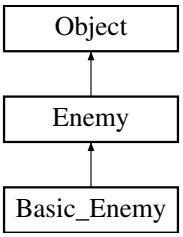
Here is a list of all documented files with brief descriptions:

Chapter 6

Class Documentation

6.1 Basic_Enemy Class Reference

Inheritance diagram for Basic_Enemy:



Public Member Functions

- **Basic_Enemy** (Level &level, Vector2D &position, int health=20, int damage=5, int range=100, int attack_↩ speed=1, int type=ObjectTypes::NoobDemon_CanAttack, int speed=20, int defense=5)
- bool attack ()

Public Member Functions inherited from Enemy

- **Enemy** (Level &level, Vector2D &position, int health, int damage, int range, int attack_speed, int type, int speed, int defense)
- int **get_speed** () const
- int **get_defense** () const
- void **move** ()
- std::vector< Vector2D > **get_route** () const
- void **set_route_position** (Vector2D position)
- Vector2D **get_prev_pos** ()
- void **set_prev_pos** (Vector2D pos)

Public Member Functions inherited from Object

- **Object** (Level &level, Vector2D &position, int health, int damage, int range, int attack_speed, int type)
- int **get_damage** () const
- int **get_health** () const
- int **get_range** () const
- int **get_attack_speed** () const
- const Vector2D **get_position** () const
- int **get_type** () const
- Level & **get_level_reference** () const
- void **set_position** (const Vector2D &position)
- void **gain_damage** (int amount)
- void **gain_health** (int amount)
- void **gain_range** (int amount)
- void **gain_attack_speed** (int amount)
- double **distance_to** (const Vector2D &target_position)
- void **lose_health** (int amount)
- State **get_state** ()
- void **set_state** (State state)

6.1.1 Member Function Documentation

6.1.1.1 attack()

```
bool Basic_Enemy::attack ( )  [virtual]
```

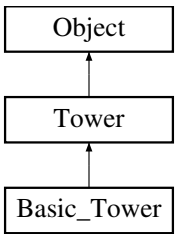
Reimplemented from Object.

The documentation for this class was generated from the following files:

- src/basic_enemy.hpp
- src/basic_enemy.cpp

6.2 Basic_Tower Class Reference

Inheritance diagram for Basic_Tower:



Public Member Functions

- **Basic_Tower** (Level &current_level, Vector2D &position, int health=30, int damage=10, int range=100, int attack_speed=1, int type=ObjectTypes::ArcherTower, int price=100, int level=1, bool attack_type_single=true)
- bool attack ()
- void **set_multiple_target** ()

Public Member Functions inherited from Tower

- **Tower** (Level &current_level, Vector2D &position, int health, int damage, int range, int attack_speed, int type, int price, int level)
- void **level_up** ()
- int **get_price** ()

Public Member Functions inherited from Object

- **Object** (Level &level, Vector2D &position, int health, int damage, int range, int attack_speed, int type)
- int **get_damage** () const
- int **get_health** () const
- int **get_range** () const
- int **get_attack_speed** () const
- const Vector2D **get_position** () const
- int **get_type** () const
- Level & **get_level_reference** () const
- void **set_position** (const Vector2D &position)
- void **gain_damage** (int amount)
- void **gain_health** (int amount)
- void **gain_range** (int amount)
- void **gain_attack_speed** (int amount)
- double **distance_to** (const Vector2D &target_position)
- void **lose_health** (int amount)
- State **get_state** ()
- void **set_state** (State state)

6.2.1 Member Function Documentation

6.2.1.1 attack()

```
bool Basic_Tower::attack ( )  [virtual]
```

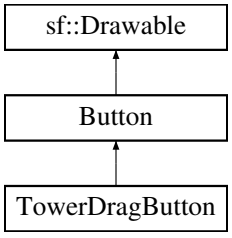
Reimplemented from Object.

The documentation for this class was generated from the following files:

- src/basic_tower.hpp
- src/basic_tower.cpp

6.3 Button Class Reference

Inheritance diagram for Button:



Public Member Functions

- **Button** (const std::string &label, sf::Vector2f size, sf::Vector2f position, sf::Color color)
- void **set_font** (sf::Font &font)
- void **set_position_text_up** (sf::Vector2f pos)
- void **set_position_text_middle** (sf::Vector2f pos)
- void **set_position_text_down** (sf::Vector2f pos)
- void **set_color** (sf::Color color)
- void **set_text_string** (const std::string &label)
- bool **is_mouse_over** (sf::RenderWindow &window)
- virtual void **handle_events** (sf::RenderWindow &window, const sf::Event &event, Level &lv)
- virtual void **some_action_from_level** (Level &lv)

Protected Member Functions

- virtual void **draw** (sf::RenderTarget &target, sf::RenderStates) const

Protected Attributes

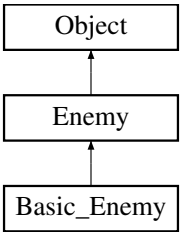
- sf::RectangleShape **_button**
- sf::Color **_button_color**
- sf::Vector2f **_position**
- sf::Vector2f **_size**
- sf::Text **_text**

The documentation for this class was generated from the following files:

- src/button.hpp
- src/button.cpp

6.4 Enemy Class Reference

Inheritance diagram for Enemy:



Public Member Functions

- **Enemy** (Level &level, Vector2D &position, int health, int damage, int range, int attack_speed, int type, int speed, int defense)
- int **get_speed** () const
- int **get_defense** () const
- void **move** ()
- std::vector< Vector2D > **get_route** () const
- void **set_route_position** (Vector2D position)
- Vector2D **get_prev_pos** ()
- void **set_prev_pos** (Vector2D pos)

Public Member Functions inherited from Object

- **Object** (Level &level, Vector2D &position, int health, int damage, int range, int attack_speed, int type)
- int **get_damage** () const
- int **get_health** () const
- int **get_range** () const
- int **get_attack_speed** () const
- const Vector2D **get_position** () const
- int **get_type** () const
- Level & **get_level_reference** () const
- void **set_position** (const Vector2D &position)
- void **gain_damage** (int amount)
- void **gain_health** (int amount)
- void **gain_range** (int amount)
- void **gain_attack_speed** (int amount)
- double **distance_to** (const Vector2D &target_position)
- void **lose_health** (int amount)
- State **get_state** ()
- void **set_state** (State state)
- virtual bool **attack** ()

The documentation for this class was generated from the following files:

- src/enemy.hpp
- src/enemy.cpp

6.5 Game Class Reference

Public Member Functions

- **Game** (const Game &)=delete
- Game **operator=** (const Game &)=delete
- int **get_side_bar_width** () const
- int **get_game_resolution** () const
- void **run** ()

The documentation for this class was generated from the following files:

- src/game.hpp
- src/game.cpp

6.6 Level Class Reference

Class that controls level of the game.

```
#include <level.hpp>
```

Public Member Functions

- Level (int resolution, int cash, int lives)

  Construct a new Level object.
- ~**Level** ()

  Destroy the Level object.
- int get_round () const

  Get current round.
- int get_cash () const

  Get current money situation.
- int get_lives () const

  Get current lives.
- std::vector< std::vector< Square * > > get_grid () const

  Get current the grid.
- int get_square_size () const

  Get the size of each square.
- void **make_grid** ()

  Makes new grid.
- void **plus_round** ()
- void add_cash (int how_much)

  Add money.
- void take_cash (int how_much)

  Take money from player.
- void take_lives (int how_much)

  Take lives from player.
- void add_lives (int how_much)

  Add lives.
- std::vector< Enemy * > get_enemies () const

  Get the vector of enemies.
- bool add_enemy (Enemy *enemy)

  Adds pointer of enemy to vector of all enemies.
- bool remove_enemy (Enemy *enemy)

  Removes enemy from vector.
- bool add_enemy_by_type (int type, Vector2D pos)

  Makes and adds new enemy by type and position of enemy.
- std::vector< Tower * > get_towers () const

  Get the vector of towers.
- bool add_tower (Tower *tower)

  Adds tower to vector of all towers.
- bool remove_tower (Tower *tower)

  Removes tower from vector of all towers.
- bool add_tower_by_type (int type, Vector2D pos)

  Makes and adds new tower to vector of all towers.
- std::pair< int, int > current_row_col (Object *obj)

  Returns current column and row of object.
- Square * current_square (Object *obj)

  Returns pointer to current square of object.
- Square * get_square_by_pos (Vector2D pos)

  Returns pointer to square by position.
- std::vector< Direction > next_road (Enemy *enemy)

  Returns vector, where road continues from current square.

- void **print_objects** ()

  Print all objects.
- int read_file (const std::string &file_name)

  Load level from file.
- int save_to_file (const std::string &file_name)

  Saves current level to file.
- void **print_map** ()

  Prints out current map.
- std::pair< int, int > can_go_notstart (Direction dir, std::vector< Direction > prev_dirs, int row, int col, bool can_go_left)

  Helper functions for randomly generate Handels situation where it's not few of first round.
- std::pair< int, int > can_go_start (Direction dir, std::vector< Direction > dir_list, int row, int col)

  Helper functions for randomly generate Handels situation where its first few moves.
- bool randomly_generate ()

  Creates fully random level.
- Square * get_first_road ()

  Returns first peace of the road Helps enemies to spawn in right place.

6.6.1 Detailed Description

Class that controls level of the game.

Class holds current grid, all enemies and tower. Also it keeps track of round, money and live situation Level can load hard coded maps from files, save current levels to files or randomly generate fully new one

Parameters

resolution	of window
cash	starting cash
lives	starting lives

6.6.2 Constructor & Destructor Documentation

6.6.2.1 Level()

```
Level::Level (
            int resolution,
            int cash,
            int lives )
```

Construct a new Level object.

Parameters

resolution	
cash	
lives	

6.6.3 Member Function Documentation

6.6.3.1 add_cash()

```
void Level::add_cash (
            int how_much )
```

Add money.

Parameters

how_much	How much money want to be added

6.6.3.2 add_enemy()

```
bool Level::add_enemy (
            Enemy * enemy )
```

Adds pointer of enemy to vector of all enemies.

Parameters

enemy	Pointer to enemy

Returns

true if is added

false if not added

6.6.3.3 add_enemy_by_type()

```
bool Level::add_enemy_by_type (
            int type,
            Vector2D pos )
```

Makes and adds new enemy by type and position of enemy.

Parameters

type	type of enemy
pos	position of enemy

Returns

true if added

false if not added

6.6.3.4 add_lives()

```
void Level::add_lives (
            int how_much )
```

Add lives.

Parameters

how_much	How much lives is added

6.6.3.5 add_tower()

```
bool Level::add_tower (
            Tower * tower )
```

Adds tower to vector of all towers.

Parameters

tower	Pointer to tower

Returns

true if added

false if not added

6.6.3.6 add_tower_by_type()

```
bool Level::add_tower_by_type (
            int type,
            Vector2D pos )
```

Makes and adds new tower to vector of all towers.

Parameters

type	type of tower
pos	position of tower

Returns

true if added

false if not added

6.6.3.7 can_go_notstart()

```
std::pair< int, int > Level::can_go_notstart (
            Direction dir,
```

```
            std::vector< Direction > prev_dirs,
            int row,
            int col,
            bool can_go_left )
```

Helper functions for randomly generate Handels situation where it's not few of first round.

Parameters

dir	Direction where randomly generate wants to go
prev_dirs	Vector of all previous directions
row	Current row
col	Current col
can_go_left	Restricts how many time function can go left

Returns

std::pair<int, int> Returns pair of next row and column

6.6.3.8 can_go_start()

```
std::pair< int, int > Level::can_go_start (
            Direction dir,
            std::vector< Direction > dir_list,
            int row,
            int col )
```

Helper functions for randomly generate Handels situation where its first few moves.

Parameters

dir	Direction where randomly generate wants to go
prev_dirs	Vector of all previous directions
row	Current row
col	Current col

Returns

std::pair<int, int> Returns pair of next row and column

6.6.3.9 current_row_col()

```
std::pair< int, int > Level::current_row_col (
            Object * obj )
```

Returns current column and row of object.

Parameters

obj	Pointer to object

Returns

std::pair<int, int> => <col, row>

6.6.3.10 current_square()

```
Square * Level::current_square (
            Object * obj )
```

Returns pointer to current square of object.

Parameters

obj	Pointer to object

Returns

Pointer to square

6.6.3.11 get_cash()

```
int Level::get_cash ( ) const
```

Get current money situation.

Returns

int

6.6.3.12 get_enemies()

```
std::vector< Enemy * > Level::get_enemies ( ) const
```

Get the vector of enemies.

Returns

std::vector<Enemy*>

6.6.3.13 get_first_road()

```
Square * Level::get_first_road ( )
```

Returns first peace of the road Helps enemies to spawn in right place.

Returns

Square∗ Pointer of square where first road is

6.6.3.14 get_grid()

```
std::vector< std::vector< Square * > > Level::get_grid ( ) const
```

Get current the grid.

Returns

std::vector<std::vector<Square∗>>

6.6.3.15 get_lives()

```
int Level::get_lives ( ) const
```

Get current lives.

Returns

int

6.6.3.16 get_round()

```
int Level::get_round ( ) const
```

Get current round.

Returns

int

6.6.3.17 get_square_by_pos()

```
Square * Level::get_square_by_pos (
            Vector2D pos )
```

Returns pointer to square by position.

Parameters

pos	Position on map

Returns

Pointer to square

6.6.3.18 get_square_size()

```
int Level::get_square_size ( ) const
```

Get the size of each square.

Returns

int

6.6.3.19 get_towers()

```
std::vector< Tower * > Level::get_towers ( ) const
```

Get the vector of towers.

Returns

std::vector<Tower*>

6.6.3.20 next_road()

```
std::vector< Direction > Level::next_road (
            Enemy * enemy )
```

Returns vector, where road continues from current square.

Parameters

enemy	Pointer to enemy

Returns

Vector of all directions where enemy can go

6.6.3.21 randomly_generate()

```
bool Level::randomly_generate ( )
```

Creates fully random level.

Returns

true If it was successful

false If it wasn't

6.6.3.22 read_file()

```
int Level::read_file (
            const std::string & file_name )
```

Load level from file.

Parameters

file_name	Name of file from where map is loaded

Returns

int 1 if maps is loaded and -1 if load failed

6.6.3.23 remove_enemy()

```
bool Level::remove_enemy (
            Enemy * enemy )
```

Removes enemy from vector.

Parameters

enemy	pointer to enemy

Returns

true if removed

false if not removes

6.6.3.24 remove_tower()

```
bool Level::remove_tower (
            Tower * tower )
```

Removes tower from vector of all towers.

Parameters

tower	Pointer to tower

Returns

true if added

false if not added

6.6.3.25 save_to_file()

```
int Level::save_to_file (
            const std::string & file_name )
```

Saves current level to file.

Parameters

file_name	Name of file where map is saved

Returns

int 1 if map is saved and -1 if failed

6.6.3.26 take_cash()

```
void Level::take_cash (
            int how_much )
```

Take money from player.

Parameters

how_much	How much money is taken

6.6.3.27 take_lives()

```
void Level::take_lives (
            int how_much )
```

Take lives from player.

Parameters

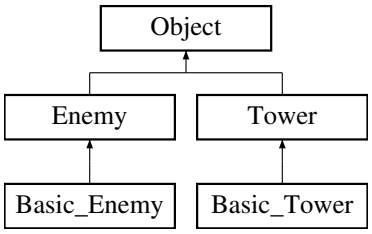
how_much	How much lives is taken

The documentation for this class was generated from the following files:

- src/level.hpp
- src/level.cpp

6.7 Object Class Reference

Inheritance diagram for Object:



Public Member Functions

- **Object** (Level &level, Vector2D &position, int health, int damage, int range, int attack_speed, int type)
- int **get_damage** () const
- int **get_health** () const
- int **get_range** () const
- int **get_attack_speed** () const
- const Vector2D **get_position** () const
- int **get_type** () const
- Level & **get_level_reference** () const
- void **set_position** (const Vector2D &position)
- void **gain_damage** (int amount)
- void **gain_health** (int amount)
- void **gain_range** (int amount)
- void **gain_attack_speed** (int amount)
- double **distance_to** (const Vector2D &target_position)
- void **lose_health** (int amount)
- State **get_state** ()
- void **set_state** (State state)
- virtual bool **attack** ()

The documentation for this class was generated from the following files:

- src/object.hpp
- src/object.cpp

6.8 Renderer Class Reference

Public Member Functions

- **Renderer** (const Renderer &)=delete
- Renderer **operator=** (const Renderer &)=delete
- void **make_drawable_level** (Level &lv)
- void **make_level_info_texts** (int game_resolution, int side_bar_width)
- void **draw_level** (sf::RenderWindow &rwindow)
- void **draw_enemy** (sf::RenderWindow &rwindow, Enemy *e_ptr, int frame)
- void **draw_enemies** (sf::RenderWindow &rwindow, std::vector< Enemy * > enemies, int frame)
- void **draw_tower** (sf::RenderWindow &rwindow, Tower *t_ptr, int frame)
- void **draw_towers** (sf::RenderWindow &rwindow, std::vector< Tower * > towers, int frame)
- void **draw_cash** (sf::RenderWindow &rwindow, int cash)
- void **draw_lives** (sf::RenderWindow &rwindow, int lives)
- void **draw_round_count** (sf::RenderWindow &rwindow, int round_count)
- void **load_font** ()

The documentation for this class was generated from the following files:

- src/renderer.hpp
- src/renderer.cpp

6.9 ResourceHandler Class Reference

Public Member Functions

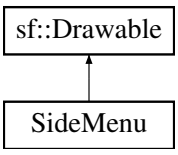
- sf::Texture & **get_texture_tower** (int type)
- sf::Texture & **get_texture_enemy** (int type)
- sf::Texture & **get_texture_tile** (int type)
- sf::Font & **get_font** ()

The documentation for this class was generated from the following files:

- src/resource_handler.hpp
- src/resource_handler.cpp

6.10 SideMenu Class Reference

Inheritance diagram for SideMenu:



Public Member Functions

- **SideMenu** (float game_resolution, float sidebar_width, ResourceHandler &rh, Level &level)
- void **update_displays** ()
- void **handle_events** (sf::RenderWindow &window, const sf::Event &event)

The documentation for this class was generated from the following files:

- src/side_menu.hpp
- src/side_menu.cpp

6.11 Square Class Reference

Public Member Functions

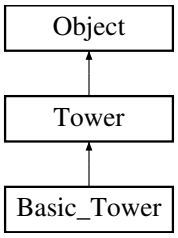
- **Square** (Vector2D center)
- Vector2D **get_center** () const
- int **get_occupied** () const
- void **print_info** ()
- bool **occupy_by_grass** ()
- bool **occupy_by_road** ()
- bool **occupy_by_tower** ()

The documentation for this class was generated from the following files:

- src/square.hpp
- src/square.cpp

6.12 Tower Class Reference

Inheritance diagram for Tower:



Public Member Functions

- **Tower** (Level &current_level, Vector2D &position, int health, int damage, int range, int attack_speed, int type, int price, int level)
- void **level_up** ()
- int **get_price** ()

Public Member Functions inherited from Object

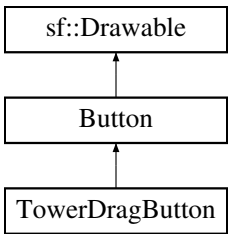
- **Object** (Level &level, Vector2D &position, int health, int damage, int range, int attack_speed, int type)
- int **get_damage** () const
- int **get_health** () const
- int **get_range** () const
- int **get_attack_speed** () const
- const Vector2D **get_position** () const
- int **get_type** () const
- Level & **get_level_reference** () const
- void **set_position** (const Vector2D &position)
- void **gain_damage** (int amount)
- void **gain_health** (int amount)
- void **gain_range** (int amount)
- void **gain_attack_speed** (int amount)
- double **distance_to** (const Vector2D &target_position)
- void **lose_health** (int amount)
- State **get_state** ()
- void **set_state** (State state)
- virtual bool **attack** ()

The documentation for this class was generated from the following files:

- src/tower.hpp
- src/tower.cpp

6.13 TowerDragButton Class Reference

Inheritance diagram for TowerDragButton:



Public Member Functions

- **TowerDragButton** (const std::string &price, sf::Vector2f size, sf::Vector2f position, sf::Color color, sf::Texture obj_texture, int tower_type)
- void **set_drag_flag** ()
- void **reset_drag_flag** ()
- bool **get_drag_flag** () const
- void **set_dragging_drawable_offset** (sf::RenderWindow &window)
- void **set_dragging_drawable_pos** (sf::RenderWindow &window)
- virtual void some_action_from_level (Level &lv)
- void handle_events (sf::RenderWindow &window, const sf::Event &event, Level &lv)

Public Member Functions inherited from Button

- **Button** (const std::string &label, sf::Vector2f size, sf::Vector2f position, sf::Color color)
- void **set_font** (sf::Font &font)
- void **set_position_text_up** (sf::Vector2f pos)
- void **set_position_text_middle** (sf::Vector2f pos)
- void **set_position_text_down** (sf::Vector2f pos)
- void **set_color** (sf::Color color)
- void **set_text_string** (const std::string &label)
- bool **is_mouse_over** (sf::RenderWindow &window)

Protected Member Functions

- virtual void draw (sf::RenderTarget &target, sf::RenderStates) const

Protected Attributes

- sf::Texture **_texture**
- sf::Sprite **_drawable_tower**
- float **_scale** = 0.05
- sf::Sprite **_drawable_dragging_tower**
- sf::Vector2f **_dragging_tower_offset**
- sf::Vector2f **_release_pos**
- int **_tower_type**
- bool **_drag_flag**

Protected Attributes inherited from Button

- sf::RectangleShape **_button**
- sf::Color **_button_color**
- sf::Vector2f **_position**
- sf::Vector2f **_size**
- sf::Text **_text**

6.13.1 Member Function Documentation

6.13.1.1 draw()

```
void TowerDragButton::draw (
            sf::RenderTarget & target,
            sf::RenderStates  ) const  [protected], [virtual]
```

Reimplemented from Button.

6.13.1.2 handle_events()

```
void TowerDragButton::handle_events (
            sf::RenderWindow & window,
            const sf::Event & event,
            Level & lv )  [virtual]
```

Reimplemented from Button.

6.13.1.3 some_action_from_level()

```
virtual void TowerDragButton::some_action_from_level (
            Level & lv )  [inline], [virtual]
```

Reimplemented from Button.

The documentation for this class was generated from the following files:

- src/tower_drag_button.hpp
- src/tower_drag_button.cpp

6.14 Vector2D Class Reference

Public Member Functions

- **Vector2D** (int x, int y)
- bool **operator==** (const Vector2D &other) const
- bool **operator!=** (const Vector2D &other) const

Public Attributes

- int **x**
- int **y**

Friends

- std::ostream & **operator**<< (std::ostream &os, const Vector2D &vec)

The documentation for this class was generated from the following file:

- src/vector2d.hpp

Chapter 7

File Documentation

7.1 attack_types.hpp

```
00001 #ifndef ATTACK_TYPES_HPP
00002 #define ATTACK_TYPES_HPP
00003
00004 #include <string>
00005 #include <iostream>
00006
00007 // #include "enemy.hpp"
00008 // #include "tower.hpp"
00009
00010 class Tower;
00011 class Enemy;
00012
00013
00014 // TODO: WaterMage and MudMage
00015 namespace ObjectTypes
00016 {
00017     enum Enemies{
00018         NoobSkeleton_NoAttack,
00019         NoobDemon_CanAttack,
00020         FastBoy,
00021         FogMage,
00022         HealerPriest,
00023         InfernoMage,
00024         TankOrc,
00025         BossKnight,
00026     };
00027     enum Towers{
00028         AoeTower,
00029         ArcherTower,
00030         MudMageTower,
00031         RepelMageTower,
00032         SniperTower,
00033         WaterMageTower,
00034     };
00035 }
00036
00037 double check_type_multiplier(Tower* tower, Enemy* enemy);
00038
00039
00040 #endif
```

7.2 basic_enemy.hpp

```
00001 #ifndef BASIC_ENEMY_HPP
00002 #define BASIC_ENEMY_HPP
00003
00004 #include "enemy.hpp"
00005 #include "attack_types.hpp"
00006 // #include "level.hpp"
00007
00008 class Basic_Enemy: public Enemy {
00009 public:
00010
```

```
00011     Basic_Enemy(Level& level, Vector2D& position, int health = 20, int damage = 5, int range = 100,
      int attack_speed = 1, int type = ObjectTypes::NoobDemon_CanAttack, int speed = 20, int defense = 5);
00012
00013     ~Basic_Enemy() { }
00014
00015     bool attack();
00016
00017     // void move();
00018 };
00019
00020 #endif
```

7.3 basic_tower.hpp

```
00001 #ifndef BASIC_TOWER_HPP
00002 #define BASIC_TOWER_HPP
00003
00004 #include "tower.hpp"
00005 #include "attack_types.hpp"
00006 #include "level.hpp"
00007
00008 class Basic_Tower: public Tower {
00009 public:
00010
00011     Basic_Tower(Level& current_level, Vector2D& position, int health = 30, int damage = 10, int range
      = 100,
00012         int attack_speed = 1, int type = ObjectTypes::ArcherTower, int price = 100, int level = 1,
      bool attack_type_single = true);
00013
00014     ~Basic_Tower() { }
00015
00016     bool attack();
00017
00018     void set_multiple_target();
00019
00020 private:
00021     bool _attack_type_single;
00022 };
00023
00024 #endif
```

7.4 button.hpp

```
00001 #ifndef TOWER_DEFENCE_SRC_BUTTON
00002 #define TOWER_DEFENCE_SRC_BUTTON
00003
00004 #include <SFML/Window.hpp>
00005 #include <SFML/Graphics.hpp>
00006 #include "level.hpp"
00007 #include <iostream>
00008
00009
00010 class Button : public sf::Drawable{
00011
00012 public:
00013     Button(){}
00014     Button(const std::string& label, sf::Vector2f size, sf::Vector2f position, sf::Color color);
00015
00016     void set_font(sf::Font& font);
00017
00018     // set the position of the button so that the text is in one of:
00019     void set_position_text_up(sf::Vector2f pos);
00020     void set_position_text_middle(sf::Vector2f pos);
00021     void set_position_text_down(sf::Vector2f pos);
00022
00023     void set_color(sf::Color color);
00024     void set_text_string(const std::string& label);
00025
00026     bool is_mouse_over(sf::RenderWindow& window);
00027
00028     virtual void handle_events(sf::RenderWindow& window, const sf::Event& event, Level& lv);
00029
00030     // TODO: define in child class
00031     virtual void some_action_from_level(Level &lv){}
00032
00033
00034 protected:
00035
00036     virtual void draw(sf::RenderTarget& target, sf::RenderStates) const;
```

```
00037
00038     sf::RectangleShape _button;
00039     sf::Color _button_color;
00040
00041     sf::Vector2f _position;
00042     sf::Vector2f _size;
00043     sf::Text _text;
00044
00045 };
00046
00047
00048
00049
00050 #endif
```

7.5 enemy.hpp

```
00001 #ifndef ENEMY_HPP
00002 #define ENEMY_HPP
00003
00004 #include "object.hpp"
00005
00006 class Enemy: public Object {
00007 public:
00008     Enemy(Level& level, Vector2D& position, int health, int damage, int range, int attack_speed, int
      type, int speed, int defense);
00009
00010     ~Enemy() { }
00011
00012     int get_speed() const;
00013
00014     int get_defense() const;
00015
00016     void move();
00017
00018     // void State get_state();
00019
00020     // bool attack();
00021
00022     std::vector<Vector2D> get_route() const;
00023
00024     void set_route_position(Vector2D position);
00025
00026     Vector2D get_prev_pos();
00027
00028     void set_prev_pos(Vector2D pos);
00029
00030 private:
00031     Vector2D _prev_pos;
00032     int _speed;
00033     int _defense;
00034     std::vector<Vector2D> _route;
00035 };
00036
00037 #endif
```

7.6 game.hpp

```
00001 #ifndef TOWER_DEFENCE_SRC_GAME
00002 #define TOWER_DEFENCE_SRC_GAME
00003
00004 #include <level.hpp>
00005 #include <SFML/Window.hpp>
00006 #include <SFML/Graphics.hpp>
00007 #include <iostream>
00008 #include <renderer.hpp>
00009 #include <random>
00010
00011 #include "basic_enemy.hpp"
00012 #include "basic_tower.hpp"
00013
00014 namespace LevelSelection{
00015     enum Choice: int{
00016         random, load
00017     };
00018 }
00019
00020
00021 /*
```

```
00022
00023 A class for running the game. Opens a window in which a game loop handles user input key, updates game
      state and draws game entities.
00024 Currently only draws place holder game entities.
00025
00026 check SFML Game Development.pdf from google.
00027
00028 */
00029
00030 class Game{
00031 public:
00032
00033     Game();
00034     ~Game(){}
00035     Game(const Game& ) = delete;
00036     Game operator=(const Game&) = delete;
00037
00038     int get_side_bar_width() const;
00039     int get_game_resolution() const;
00040
00041     // call from main, runs the game loop until a the window is closed,
00042     void run();
00043
00044
00045
00046 private:
00047
00048     // makes the grid for the level and then fills it according to the chosen style: random or load
      from file
00049     int generate_chosen_level_style(int chosen_lv);
00050
00051     // open window
00052     void open_window();
00053
00054     // process events in loop:  clicks, button presses
00055     void process_events();
00056
00057     // update game state in loop: attacks, movement
00058     void update();
00059
00060     // draws a frame in loop,
00061     void render();
00062
00063     void start_round();
00064
00065     void update_enemies();
00066
00067     void update_towers();
00068
00069     // for testing only
00070     Vector2D some_pos;
00071
00072     int _game_resolution;
00073     int _side_bar_width;
00074
00075     sf::RenderWindow _window;
00076
00077     Renderer _renderer;
00078     Level _level;
00079
00080     bool round_over = false;
00081 };
00082 #endif
```

7.7 level.hpp

```
00001 #ifndef Level_HPP
00002 #define Level_HPP
00003
00004 #include "square.hpp"
00005 #include "vector2d.hpp"
00006 #include "object.hpp"
00007
00008 #include <vector>
00009 #include <string>
00010 #include <fstream>
00011 #include <iostream>
00012 #include <sstream>
00013 #include <cstdlib>
00014 #include <ctime>
00015 #include <cmath>
00016
00017
```

```
00024 enum Direction{
00025     up, down, right, left
00026 };
00027
00028
00039 class Level {
00040 public:
00048     Level(int resolution, int cash, int lives);
00049
00053     ~Level() {
00054         for (std::vector<Square*>& column : _grid){
00055             for (Square* s : column){
00056                 delete s;
00057             }
00058             column.clear();
00059         }
00060         _grid.clear();
00061
00062         for (auto* e : _enemies){
00063             delete e;
00064         }
00065         _enemies.clear();
00066
00067         for (auto* t : _towers){
00068             delete t;
00069         }
00070         _towers.clear();
00071     }
00072
00077     int get_round() const;
00078
00083     int get_cash() const;
00084
00089     int get_lives() const;
00090
00095     std::vector<std::vector<Square*» get_grid() const;
00096
00101     int get_square_size() const;
00102
00106     void make_grid();
00107
00111     void plus_round();
00112
00117     void add_cash(int how_much);
00118
00123     void take_cash(int how_much);
00124
00129     void take_lives(int how_much);
00130
00135     void add_lives(int how_much);
00136
00141     std::vector<Enemy*> get_enemies() const;
00142
00149     bool add_enemy(Enemy* enemy);
00150
00157     bool remove_enemy(Enemy* enemy);
00158
00167     bool add_enemy_by_type(int type, Vector2D pos);
00168
00173     std::vector<Tower*> get_towers() const;
00174
00181     bool add_tower(Tower* tower);
00182
00189     bool remove_tower(Tower* tower);
00190
00198     bool add_tower_by_type(int type, Vector2D pos);
00199
00200     // returns current column and row of object
00206     std::pair<int, int> current_row_col(Object* obj);
00207
00213     Square* current_square(Object* obj);
00214
00220     Square* get_square_by_pos(Vector2D pos);
00221
00227     std::vector<Direction> next_road(Enemy* enemy);
00228
00232     void print_objects();
00233
00239     int read_file(const std::string& file_name);
00240
00246     int save_to_file(const std::string& file_name);
00247
00251     void print_map();
00252
00253     // Helper functions for randomly generate
00254     // One handels situations of first few moves and other all the rest ones
00265     std::pair<int, int> can_go_notstart(Direction dir, std::vector<Direction> prev_dirs, int row, int
```

```
    col, bool can_go_left);
00266
00276     std::pair<int, int> can_go_start(Direction dir, std::vector<Direction> dir_list, int row, int
    col);
00277
00283     bool randomly_generate();
00284
00290     Square* get_first_road();
00291
00292 private:
00293     Square* _first_road;
00294     int _square_size;
00295     int _round, _cash, _lives;
00296     std::vector<std::vector<Square*>> _grid;
00297     std::vector<Enemy*> _enemies;
00298     std::vector<Tower*> _towers;
00299 };
00300
00301 #endif
```

7.8 object.hpp

```
00001 #ifndef OBJECT_HPP
00002 #define OBJECT_HPP
00003
00004 #include "vector2d.hpp"
00005
00006 #include <vector>
00007 #include <math.h>
00008 #include <algorithm>
00009 #include <stdexcept>
00010 #include <chrono>
00011 #include <thread>
00012
00013 class Level;
00014
00015 enum State{none, walking_right, walking_left, attacking_right, attacking_left, dying};
00016
00017 class Object {
00018 public:
00019     Object(Level& level, Vector2D& position, int health, int damage, int range, int attack_speed, int
    type);
00020
00021     virtual ~Object();
00022
00023     int get_damage() const;
00024     int get_health() const;
00025     int get_range() const;
00026     int get_attack_speed() const;
00027     const Vector2D get_position() const;
00028     int get_type() const;
00029     Level& get_level_reference() const;
00030
00031     void set_position(const Vector2D& position);
00032
00033     void gain_damage(int amount);
00034     void gain_health(int amount);
00035     void gain_range(int amount);
00036     void gain_attack_speed(int amount);
00037
00038     double distance_to(const Vector2D& target_position);
00039
00040     void lose_health(int amount);
00041
00042     State get_state();
00043
00044     void set_state(State state);
00045
00046     virtual bool attack();
00047
00048 private:
00049     Level& _level;
00050     int _health_points;
00051     int _damage;
00052     int _range;
00053     int _attack_speed;
00054     Vector2D _position;
00055     int _type;
00056     State _state;
00057 };
00058
00059 #endif
```

7.9 renderer.hpp

```
00001 #ifndef TOWER_DEFENCE_SRC_RENDERER_HPP
00002 #define TOWER_DEFENCE_SRC_RENDERER_HPP
00003
00004
00005 #include "vector2d.hpp"
00006 #include "level.hpp"
00007 #include "object.hpp"
00008 #include <SFML/Window.hpp>
00009 #include <SFML/Graphics.hpp>
00010 #include <iostream>
00011 #include <string>
00012 #include "resource_handler.hpp"
00013
00014 #include "attack_types.hpp"
00015
00016
00017 /* Class for creating drawable game objects for window.draw([DRAWABLE GAME OBJECT]) .*/
00018 /* TODO: Make another class for loading and storing textures, in some data structure, for example 2d
      array or similar:
00019     texture = Textures.get_texture(object_type, object_state), where on object_state to choose between
      shoot left/right, die etc.*/
00020
00021 class Renderer{
00022 public:
00023
00024     Renderer();
00025     ~Renderer(){}
00026     Renderer(const Renderer& ) = delete;
00027     Renderer operator=(const Renderer&) = delete;
00028
00029     // call at the beginning of the game
00030     void make_drawable_level(Level & lv);
00031     void make_level_info_texts(int game_resolution, int side_bar_width);
00032
00033     //void make_drawable_buttons()
00034
00035     // draw background
00036     void draw_level(sf::RenderWindow& rwindow);
00037
00038     // draw single enemy
00039     void draw_enemy(sf::RenderWindow& rwindow, Enemy* e_ptr, int frame); // TODO: implement the choice
      of correct texture with obn and object type
00040
00041     // draw enemies on from a list
00042     void draw_enemies(sf::RenderWindow& rwindow, std::vector< Enemy * > enemies, int frame); // TODO:
      remove last argument with real textures
00043
00044     // draw single enemy
00045     void draw_tower(sf::RenderWindow& rwindow, Tower* t_ptr, int frame); // TODO: implement the choice
      of correct texture with object state and object type
00046
00047     // draw towers on from a list
00048     void draw_towers(sf::RenderWindow& rwindow, std::vector< Tower * > towers, int frame); // TODO:
      remove last argument with real textures
00049
00050
00051     // draw texts
00052     void draw_cash(sf::RenderWindow& rwindow, int cash);
00053     void draw_lives(sf::RenderWindow& rwindow, int lives);
00054     void draw_round_count(sf::RenderWindow& rwindow, int round_count);
00055
00056     void load_font();
00057
00058 private:
00059     // a sprite for drawing grid
00060     sf::Sprite _drawable_level;
00061
00062     // a sprite for drawing objects
00063     sf::Sprite _drawable_enemy;
00064     sf::Sprite _drawable_tower;
00065
00066     // grids connected as a one RenderedTexture
00067     sf::RenderTexture _level_texture;
00068
00069     // place holders
00070     sf::RenderTexture _tower_texture;
00071     sf::RenderTexture _enemy_texture;
00072
00073     float scale_factor; // number that scales textures to right size
00074
00075     sf::Texture _tower_sprite; // for tower texture
00076     sf::Texture _enemy_sprite; // for enemy texture
00077
00078     sf::Texture _grass_pic;
00079     sf::Texture _road_pic;
```

```
00080
00081     float _scale_factor_tower = 2.5; // Adjust this value as needed
00082     float _scale_factor_enemy = 1; // TODO: some enemy type depending value
00083
00084     sf::Font _font;
00085     sf::Text _round_count_text;
00086     sf::Text _cash_text;
00087     sf::Text _lives_text;
00088
00089     // Resource handler
00090     ResourceHandler _rh;
00091 };
00092
00093 #endif
```

7.10 resource_handler.hpp

```
00001 #ifndef TOWER_DEFENCE_SRC_RESOURCE_HANDLER
00002 #define TOWER_DEFENCE_SRC_RESOURCE_HANDLER
00003
00004 #include<SFML/Graphics.hpp>
00005 #include<memory>
00006 #include "attack_types.hpp"
00007
00008 /*
00009     loads ALL textures and gives them as pointers.
00010     needs to be created in game class, pass as reference
00011 */
00012
00013 class ResourceHandler{
00014
00015 public:
00016     ResourceHandler(){load_all_textures(); load_font();}
00017
00018     // functions to get pointers to textures by type of object
00019     sf::Texture& get_texture_tower(int type);
00020     sf::Texture& get_texture_enemy(int type);
00021     sf::Texture& get_texture_tile(int type);
00022     sf::Font& get_font();
00023
00024 private:
00025
00026     void load_all_textures();
00027
00028     // load functions for all textures
00029     // someimage.png -> sf::Texture
00030     void load_texture_tower(int type, const std::string& filename);
00031     void load_texture_enemy(int type, const std::string& filename);
00032     void load_texture_tile(int type, const std::string& filename);
00033     void load_font();
00034
00035     // place holders for all textures
00036     // object_type -->  texture_ptr
00037     std::map<int, std::shared_ptr<sf::Texture>> _towers_textures_ptr_map;
00038     std::map<int, std::shared_ptr<sf::Texture>> _enemies_textures_ptr_map;
00039     std::map<int, std::shared_ptr<sf::Texture>> _tiles_textures_ptr_map;
00040     sf::Font _font;
00041 };
00042
00043
00044 #endif
```

7.11 side_menu.hpp

```
00001 #ifndef TOWER_DEFENCE_SRC_SIDEMENU
00002 #define TOWER_DEFENCE_SRC_SIDEMENU
00003
00004 #include <SFML/Window.hpp>
00005 #include <SFML/Graphics.hpp>
00006 #include "tower_drag_button.hpp"
00007 #include "level.hpp"
00008 #include "resource_handler.hpp"
00009
00010 #include "iostream"
00011
00012 class SideMenu : public sf::Drawable {
00013
00014 public:
00015
```

```
00016     SideMenu(float game_resolution, float sidebar_width, ResourceHandler& rh,   Level& level);
00017     ~SideMenu(){
00018         for (auto button : _drag_buttons){
00019             delete button;
00020         }
00021     }
00022
00023     void update_displays();
00024     void handle_events(sf::RenderWindow& window, const sf::Event& event);
00025
00026 private:
00027
00028     void setup_background();
00029     void setup_buttons();
00030     void setup_info_displays();
00031     void setup_menu_title();
00032
00033     // derived from drawable;
00034     virtual void draw( sf::RenderTarget& target, sf::RenderStates states) const;
00035
00036     //TODO: freeze buttons during gameplay?
00037     bool is_paused;
00038
00039     float _game_resolution;
00040     float _side_menu_width;
00041
00042     sf::Text _title;
00043     sf::RectangleShape _background;
00044
00045     sf::Text _round_count_text;
00046     sf::Text _cash_text;
00047     sf::Text _lives_text;
00048
00049
00050     std::vector< TowerDragButton *> _drag_buttons;
00051
00052     Level& _level;
00053
00054     // to get textures
00055     ResourceHandler _rh;
00056 };
00057
00058
00059 #endif
```

7.12 square.hpp

```
00001 #ifndef SQUARE_HPP
00002 #define SQUARE_HPP
00003
00004 #include "vector2d.hpp"
00005 #include "object.hpp"
00006 #include <iostream>
00007 #include "tower.hpp"
00008 #include "enemy.hpp"
00009
00010 #include <vector>
00011
00012 enum occupied_type{
00013     grass, road, tower
00014 };
00015
00016
00017 class Square {
00018 public:
00019     Square(Vector2D center);
00020
00021     ~Square() { }
00022
00023     // returns center cordinates of square
00024     Vector2D get_center() const;
00025
00026     // Returns what is occupuing square
00027     int get_occupied() const;
00028
00029     void print_info();
00030
00031     // Occupies square by something
00032     bool occupy_by_grass();
00033     bool occupy_by_road();
00034     bool occupy_by_tower();
00035
00036 private:
```

```
00037     Vector2D _center;
00038     occupied_type _occupied_by;
00039 };
00040
00041
00042 #endif
```

7.13 tower.hpp

```
00001 #ifndef TOWER_HPP
00002 #define TOWER_HPP
00003
00004 #include "object.hpp"
00005
00006 class Tower: public Object {
00007 public:
00008     Tower(Level& current_level, Vector2D& position, int health, int damage, int range, int
      attack_speed, int type, int price, int level);
00009
00010     ~Tower() { }
00011
00012     void level_up();
00013
00014     int get_price();
00015
00016 private:
00017     int _price;
00018     int _level;
00019 };
00020 #endif
```

7.14 tower_drag_button.hpp

```
00001 #ifndef TOWER_DEFENCE_SRC_TOWERDRAGBUTTON
00002 #define TOWER_DEFENCE_SRC_TOWERDRAGBUTTON
00003
00004 #include <SFML/Window.hpp>
00005 #include <SFML/Graphics.hpp>
00006 #include "button.hpp"
00007
00008 class TowerDragButton : public Button{
00009 public:
00010     TowerDragButton(){}
00011     TowerDragButton(const std::string& price, sf::Vector2f size, sf::Vector2f position, sf::Color
      color, sf::Texture obj_texture, int tower_type);
00012
00013     /*
00014     1. set the position of the tower to where square where the mouse was released.
00015         1.1 if grid square empty call some function on level to create new tower. TODO: handle this
      including cash, purchases in side level
00016         1.2 otherwise dont make new object and release image
00017     reset drag_flag
00018     */
00019
00020
00021     void set_drag_flag();
00022     void reset_drag_flag();
00023     bool get_drag_flag() const ;
00024
00025     // get mouse coords and place dragging image on that position
00026
00027     void set_dragging_drawable_offset(sf::RenderWindow& window);
00028
00029     void set_dragging_drawable_pos(sf::RenderWindow& window);
00030
00031     // define to create object to place in mouse release pos
00032     virtual void some_action_from_level(Level &lv){}
00033
00034     // implements the operation of the drag trough events
00035     void handle_events(sf::RenderWindow& window, const sf::Event& event, Level& lv);
00036
00037 protected:
00038
00039     /* draw the dragging_image, static button image of object, bounding square, object name text,*/
00040     virtual void draw(sf::RenderTarget& target, sf::RenderStates) const;
00041
00042
00043
00044
```

```
00045 // display object image in button.
00046 sf::Texture _texture;
00047 sf::Sprite _drawable_tower;
00048
00049 // scale image accordingly
00050 float _scale = 0.05;
00051
00052 // to show image behind mouse while dragging, use blurred image of the same image
00053 sf::Sprite _drawable_dragging_tower;
00054
00055 // offset for dragging image - reduce from mouse pos to set dragging_image pos
00056 sf::Vector2f _dragging_tower_offset;
00057 // where mouse was released -> determine grid from this
00058 sf::Vector2f _release_pos;
00059 // determine image of the button, text, what will be created what will the button represent
00060 int _tower_type;
00061 // determines whether the image is dragged or not
00062 bool _drag_flag;
00063
00064 };
00065
00066
00067
00068
00069 #endif
```

7.15 vector2d.hpp

```
00001 #ifndef VECTOR2D_HPP
00002 #define VECTOR2D_HPP
00003
00004 #include <iostream>
00005
00006 class Vector2D {
00007 public:
00008     int x;
00009     int y;
00010
00011     Vector2D() : x(0), y(0) {}
00012     Vector2D(int x, int y) : x(x), y(y) {}
00013
00014     bool operator==(const Vector2D& other) const {
00015         return (x == other.x && y == other.y);
00016     }
00017
00018     bool operator!=(const Vector2D& other) const {
00019         return !(*this == other);
00020     }
00021
00022     friend std::ostream& operator«(std::ostream& os, const Vector2D& vec) {
00023         os « vec.x « " " « vec.y;
00024         return os;
00025     }
00026 };
00027
00028 #endif
```

7.16 EnemyTests.cpp

```
00001 // #include "enemy.hpp"
00002 // #include <iostream>
00003
00004 // bool testEnemySpeed() {
00005 //     Vector2D position(0, 0);
00006 //     Level level(1000, 100, 3);
00007 //     Enemy enemy(level, 100, 20, 5, 3, position, 1, 8, 10);
00008 //     return enemy.get_speed() == 8;
00009 // }
00010
00011 // bool testEnemyDefense() {
00012 //     Vector2D position(0, 0);
00013 //     Level level(1000, 100, 3);
00014 //     Enemy enemy(level, 100, 20, 5, 3, position, 1, 8, 10);
00015 //     return enemy.get_defense() == 10;
00016 // }
00017
00018 // bool testEnemySetTargetPositionAndGetRoute() {
00019 //     Vector2D position(0, 0);
00020 //     Level level(1000, 100, 3);
```

```
00021 //     Enemy enemy(level, 100, 20, 5, 3, position, 1, 8, 10);
00022
00023 //     Vector2D targetPosition(10, 10);
00024 //     enemy.set_route_position(targetPosition);
00025 //     std::vector<Vector2D> route = enemy.get_route();
00026
00027 //     return route.size() == 2 &&
00028 //            route[0] == Vector2D(0, 0) &&
00029 //            route[1] == targetPosition &&
00030 //            enemy.get_position() == targetPosition;
00031 // }
00032
00033 // int enemy_tests() {
00034 //     int testsFailed = 0;
00035
00036 //     if (testEnemySpeed()) {
00037 //         std::cout « "testEnemySpeed: Passed" « std::endl;
00038 //     } else {
00039 //         std::cout « "testEnemySpeed: Failed" « std::endl;
00040 //         testsFailed++;
00041 //     }
00042
00043 //     if (testEnemyDefense()) {
00044 //         std::cout « "testEnemyDefense: Passed" « std::endl;
00045 //     } else {
00046 //         std::cout « "testEnemyDefense: Failed" « std::endl;
00047 //         testsFailed++;
00048 //     }
00049
00050 //     if (testEnemySetTargetPositionAndGetRoute()) {
00051 //         std::cout « "testEnemySetTargetPositionAndGetRoute: Passed" « std::endl;
00052 //     } else {
00053 //         std::cout « "testEnemySetTargetPositionAndGetRoute: Failed" « std::endl;
00054 //         testsFailed++;
00055 //     }
00056
00057 //     if (testsFailed == 0) {
00058 //         std::cout « "All tests passed." « std::endl;
00059 //     } else {
00060 //         std::cout « "Some tests failed." « std::endl;
00061 //     }
00062
00063 //     return testsFailed;
00064 // }
```

7.17 LevelTests.cpp

```
00001 // #include "level.hpp"
00002 // #include "object.hpp"
00003 // #include <iostream>
00004 // #include <cstdlib>
00005 // #include <fstream>
00006 // #include <iostream>
00007 // #include <sstream>
00008 // #include <string>
00009 // #include <vector>
00010
00011 // // tests round count
00012 // bool testRound(){
00013 //     Level lv(1000, 1000, 50); // new level
00014 //     int random_int = rand() % 10;
00015 //     for (int i = 0; i < random_int; i++) // add random amount of rounds
00016 //     {
00017 //         lv.plus_round();
00018 //     }
00019 //     return lv.get_round() == random_int; // checks if count was correct
00020 // }
00021
00022 // // tests cash count
00023 // bool testCash(){
00024 //     Level lv(1000, 1000, 50); // new level
00025 //     int random_int = rand() % 500; // add some random number of cash
00026 //     lv.add_cash(random_int);
00027 //     int random_int2 = rand() % 100; // take some random number of cash
00028 //     lv.take_cash(random_int2);
00029 //     return lv.get_cash() == (1000 + random_int - random_int2); // checks if cash count was correct
00030 // }
00031
00032 // // tests lives count
00033 // bool testLives(){
00034 //     Level lv(1000, 1000, 50); // new level
00035 //     int random_int = rand() % 25;
00036 //     lv.take_lives(random_int); // Minuses random amount of money
```

```
00037 //     int random_int2 = rand() % 10;
00038 //     lv.add_lives(random_int2); // add random amount of money
00039 //     return lv.get_lives() == (50 - random_int + random_int2); // checks if lives count
00040 // }
00041
00042 // bool testObjectList(){
00043 //     std::string file_name = "maps/example_map.txt"; // file name of the map test map
00044 //     Level lv(1000, 1000, 50); // new level
00045 //     lv.make_grid();
00046 //     if (lv.read_file(file_name) == -1){  // reads new map from test map file
00047 //         std::cout << "File reading failed" << std::endl;
00048 //         return false;
00049 //     }
00050
00051 //     Vector2D pos = Vector2D(150, 450); // should fail
00052 //     Tower* t = new Tower(lv, 10, 10, 10, 10, pos, 10, 10, 10);
00053
00054 //     Vector2D pos2 = Vector2D(50, 50); // should pass
00055 //     Tower* t2 = new Tower(lv, 10, 10, 10, 10, pos2, 10, 10, 10);
00056
00057 //     Vector2D pos3 = Vector2D(350, 350); // should fail
00058 //     Enemy* e = new Enemy(lv, 10, 10, 10, 10, pos3, 10, 1, 10);
00059
00060 //     Vector2D pos4 = Vector2D(150, 455); // should pass
00061 //     Enemy* e2 = new Enemy(lv, 10, 10, 10, 10, pos4, 10, 1, 10);
00062
00063 //     // std::cout << !lv.add_tower(t) << lv.add_tower(t2) << !lv.add_enemy(e) << lv.add_enemy(e2) <<
    std::endl;
00064
00065 //     return !lv.add_tower(t) && lv.add_tower(t2) && !lv.add_enemy(e) && lv.add_enemy(e2);
00066 // }
00067
00068 // // test that makeGrid function makes grid that is 10 x 10
00069 // bool testGridSize(){
00070 //     Level lv(1000, 1000, 50); // new level
00071 //     lv.make_grid();
00072 //     std::vector<std::vector<Square*>> grid = lv.get_grid(); // new grid
00073 //     if (grid.size() != 10){ // checks that there is 10 columns
00074 //         return false; // returns false if not
00075 //     }
00076 //     for (size_t i = 0; i < grid.size(); i++) // checks that every column have 10 squares
00077 //     {
00078 //         std::vector<Square*> column = grid[i];
00079 //         if (column.size() != 10){
00080 //             return false; // returns false if not
00081 //         }
00082 //     }
00083 //     return true;
00084 // }
00085
00086 // // checks that makeGrid function initialize squares with right center points
00087 // bool testGridSquareCenters(){
00088 //     Level lv(1000, 1000, 50); // new level
00089 //     lv.make_grid();
00090 //     std::vector<std::vector<Square*>> grid = lv.get_grid(); // new grid
00091 //     int x = 5; // cordinates for first square center
00092 //     int y = 5;
00093 //     for (size_t i = 0; i < grid.size(); i++) // checks that every square has correct center points
00094 //     {
00095 //         int current_x = x + (i * 10); // calculates what x should be
00096 //         std::vector<Square*> column = grid[i];
00097 //         for (size_t j = 0; j < column.size(); j++)
00098 //         {
00099 //             int current_y = y + (j * 10); // calculates what y should be
00100 //             Vector2D current_center(current_x, current_y); // makes correct cordinates
00101 //             if (column[j]->get_center() == current_center){ // compares if cordinates matches
00102 //                 //lv.~Level(); // deletes if not
00103 //                 return false;
00104 //             }
00105 //         }
00106 //     }
00107 //     //lv.~Level(); // deletes when test is over
00108 //     return true;
00109 // }
00110
00111 // bool testCurrentRowCol(){
00112 //     std::string file_name = "maps/example_map.txt"; // file name of the map test map
00113 //     Level lv(1000, 1000, 50); // new level
00114 //     lv.make_grid();
00115 //     if (lv.read_file(file_name) == -1){  // reads new map from test map file
00116 //         std::cout << "File reading failed" << std::endl;
00117 //         return false;
00118 //     }
00119 //     Vector2D pos = Vector2D(50, 50); // should be <0, 0>
00120 //     Tower* t = new Tower(lv, 10, 10, 10, 10, pos, 10, 10, 10);
00121
00122 //     Vector2D pos2 = Vector2D(150, 455); // should be <1, 4>
```

```
00123 //     Enemy* e = new Enemy(lv, 10, 10, 10, 10, pos2, 10, 1, 10);
00124
00125 //     // std::cout << lv.current_row_col(t).first << lv.current_row_col(t).second
00126 //     //    << lv.current_row_col(e).first << lv.current_row_col(e).second << std::endl;
00127
00128 //     return lv.current_row_col(t) == std::make_pair(0, 0) && lv.current_row_col(e) ==
      std::make_pair(1, 4);
00129 // }
00130
00131 // bool testCurrentSquare(){
00132 //     std::string file_name = "maps/example_map.txt"; // file name of the map test map
00133 //     Level lv(1000, 1000, 50); // new level
00134 //     lv.make_grid();
00135 //     if (lv.read_file(file_name) == -1){  // reads new map from test map file
00136 //         std::cout << "File reading failed" << std::endl;
00137 //         return false;
00138 //     }
00139 //     Vector2D pos = Vector2D(50, 50); // should be <0, 0>
00140 //     Tower* t = new Tower(lv, 10, 10, 10, 10, pos, 10, 10, 10);
00141
00142 //     Vector2D pos2 = Vector2D(150, 450); // should be <2, 5>
00143 //     Enemy* e = new Enemy(lv, 10, 10, 10, 10, pos2, 10, 1, 10);
00144
00145 //     std::vector<std::vector<Square*>> grid = lv.get_grid();
00146
00147 //     // std::cout << lv.current_square(t)->get_center() << " - " << grid[0][0]->get_center() << " - "
00148 //     //    << lv.current_square(e)->get_center() << " - " << grid[1][4]->get_center() << std::endl;
00149
00150 //     return lv.current_square(t)->get_center() == grid[0][0]->get_center()
00151 //         && lv.current_square(e)->get_center() == grid[1][4]->get_center();
00152 // }
00153
00154 // bool testGetSquareByPos(){
00155 //     std::string file_name = "maps/example_map.txt"; // file name of the map test map
00156 //     Level lv(1000, 1000, 50); // new level
00157 //     lv.make_grid();
00158 //     if (lv.read_file(file_name) == -1){  // reads new map from test map file
00159 //         std::cout << "File reading failed" << std::endl;
00160 //         return false;
00161 //     }
00162 //     Vector2D pos = Vector2D(50, 50); // should be <0, 0>
00163
00164 //     std::vector<std::vector<Square*>> grid = lv.get_grid();
00165
00166 //     return grid[0][0] == lv.get_square_by_pos(pos);
00167 // }
00168
00169 // bool testNextRoad(){
00170 //     std::string file_name = "maps/example_map.txt"; // file name of the map test map
00171 //     Level lv(1000, 1000, 50); // new level
00172 //     lv.make_grid();
00173 //     if (lv.read_file(file_name) == -1){  // reads new map from test map file
00174 //         std::cout << "File reading failed" << std::endl;
00175 //         return false;
00176 //     }
00177 //     Vector2D pos2 = Vector2D(150, 450); // should be <1, 4>
00178 //     Enemy* e = new Enemy(lv, 10, 10, 10, 10, pos2, 10, 1, 10);
00179
00180 //     std::vector<Direction> res = lv.next_road(e);
00181
00182 //     // std::cout << res.size() << res[0] << res[1] << std::endl;
00183
00184 //     return res[0] == right && res[1]== up && res.size() == 2;
00185 // }
00186
00187 // // Test for read and write to file
00188
00189 // bool testRead(){
00190 //     std::string file_name = "maps/example_map.txt"; // file name of the map test map
00191 //     Level lv(1000, 1000, 50); // new level
00192 //     lv.make_grid();
00193 //     if (lv.read_file(file_name) == -1){  // reads new map from test map file
00194 //         std::cout << "File reading failed" << std::endl;
00195 //         return false;
00196 //     }
00197 //     std::vector<std::vector<Square*>> grid = lv.get_grid();
00198 //     std::ifstream file(file_name);
00199 //     for (size_t i = 0; i < grid.size(); i++) // compares grid to test map file
00200 //     {
00201 //         std::string line;
00202 //         std::getline(file, line);
00203 //         std::vector<Square*> column = grid[i];
00204 //         for (size_t j = 0; j < column.size(); j++)
00205 //         {
00206 //             if (line[j] == '#' && column[j]->get_occupied() == road){
00207 //                 return false;
00208 //             }
```

```
00209 //         }
00210 //     }
00211 //     return true;
00212 // }
00213
00214 // bool testWrite(){
00215 //     std::string file_name = "maps/example_map.txt"; // file name for reading
00216 //     std::string file_name_w = "maps/example_map_w.txt"; // file name for writing
00217 //     Level lv(1000, 1000, 50); // new level
00218 //     lv.make_grid();
00219 //     lv.read_file(file_name); // reads maps from file
00220 //     lv.save_to_file(file_name_w); // writes current map to file
00221
00222 //     // compares two two files
00223 //     std::ifstream f1(file_name, std::ifstream::binary|std::ifstream::ate);
00224 //     std::ifstream f2(file_name_w, std::ifstream::binary|std::ifstream::ate);
00225
00226 //     if (f1.fail() || f2.fail()) {
00227 //         return false; //file problem
00228 //     }
00229
00230 //     if (f1.tellg() != f2.tellg()) {
00231 //         return false; //size mismatch
00232 //     }
00233
00234 //     //seek back to beginning and use std::equal to compare contents
00235 //     f1.seekg(0, std::ifstream::beg);
00236 //     f2.seekg(0, std::ifstream::beg);
00237 //     return std::equal(std::istreambuf_iterator<char>(f1.rdbuf()),
00238 //                       std::istreambuf_iterator<char>(),
00239 //                       std::istreambuf_iterator<char>(f2.rdbuf()));
00240 // }
00241
00242 // bool testRandomMap(){
00243 //     Level lv(1000, 1000, 50); // new level
00244 //     lv.make_grid();
00245 //     bool res = lv.randomly_generate();
00246 //     lv.print_map();
00247 //     return res;
00248 // }
00249
00250 // /*bool testRandomHelp(){
00251 //     Level lv(1000, 1000, 50); // new level
00252 //     lv.make_grid();
00253 //     std::vector<Direction> list;
00254 //     list.push_back(right);
00255 //     list.push_back(right);
00256 //     bool res = !lv.can_go_notfirst(right, list); // should fail
00257 //     list.push_back(left);
00258 //     res = lv.can_go_notfirst(right, list); // should pass
00259 //     list.clear();
00260 //     list.push_back(down);
00261 //     list.push_back(down);
00262 //     res = !lv.can_go_notfirst(up, list);// should fail
00263 //     list.push_back(right);
00264 //     res = lv.can_go_notfirst(up, list); // should pass
00265 //     list.clear();
00266 //     list.push_back(up);
00267 //     list.push_back(up);
00268 //     res = !lv.can_go_notfirst(down, list); // should fail
00269 //     list.push_back(right);
00270 //     res = lv.can_go_notfirst(down, list); // should pass
00271 //     list.clear();
00272 //     list.push_back(left);
00273 //     list.push_back(left);
00274 //     res = !lv.can_go_notfirst(left, list); // should fail
00275 //     list.push_back(right);
00276 //     res = lv.can_go_notfirst(left, list); // should pass
00277 //     return res;
00278 // }*/
00279
00280 // /*bool testRandom1(){ // test for can_go_start()
00281 //     Level lv(1000, 1000, 50); // new level
00282 //     lv.make_grid();
00283 //     std::vector<Direction> list;
00284 //     std::pair<int, int> pair = lv.can_go_start(right, list, 4, 10);
00285 //     std::cout « pair.first « " " « pair.second « std::endl;
00286 //     list.push_back(right);
00287 //     pair = lv.can_go_start(left, list, 4, 1);
00288 //     std::cout « pair.first « " " « pair.second « std::endl;
00289 //     list.clear();
00290 //     list.push_back(down);
00291 //     pair = lv.can_go_start(up, list, 4, 0);
00292 //     std::cout « pair.first « " " « pair.second « std::endl;
00293 //     list.clear();
00294 //     list.push_back(up);
00295 //     pair = lv.can_go_start(down, list, 4, 0);
```

```
00296 //     std::cout « pair.first « " " « pair.second « std::endl;
00297 //     return true;
00298 // }*/
00299
00300 // /*bool testRandom2(){
00301 //     Level lv(1000, 1000, 50); // new level
00302 //     lv.make_grid();
00303 //     std::vector<Direction> list;
00304 //     list.push_back(right);
00305 //     list.push_back(up);
00306 //     std::pair<int, int> pair = lv.can_go_notstart(up, list, 10, 4, true);
00307 //     std::cout « pair.first « " " « pair.second « std::endl;
00308
00309 //     list.push_back(right);
00310 //     list.push_back(down);
00311 //     pair = lv.can_go_notstart(down, list, 0, 4, true);
00312 //     std::cout « pair.first « " " « pair.second « std::endl;
00313
00314 //     list.push_back(up);
00315 //     list.push_back(up);
00316 //     pair = lv.can_go_notstart(right, list, 4, 10, true);
00317 //     std::cout « pair.first « " " « pair.second « std::endl;
00318
00319 //     list.push_back(up);
00320 //     list.push_back(up);
00321 //     pair = lv.can_go_notstart(left, list, 4, 0, true);
00322 //     std::cout « pair.first « " " « pair.second « std::endl;
00323
00324 //     return true;
00325 // }*/
00326
00327 // static int level_test(){
00328 //     srand((unsigned int)time(NULL)); // makes rand() more random
00329 //     int fails = 0;
00330
00331 //     if (testRound()){
00332 //         std::cout « "testRound: Passed" « std::endl;
00333 //     } else {
00334 //         std::cout « "testRound: Failed" « std::endl;
00335 //         fails++;
00336 //     }
00337
00338 //     if (testCash()){
00339 //         std::cout « "testCash: Passed" « std::endl;
00340 //     } else {
00341 //         std::cout « "testCash: Failed" « std::endl;
00342 //         fails++;
00343 //     }
00344
00345 //     if (testLives()){
00346 //         std::cout « "testLives: Passed" « std::endl;
00347 //     } else {
00348 //         std::cout « "testLives: Failed" « std::endl;
00349 //         fails++;
00350 //     }
00351
00352 //     if (testGridSize()){
00353 //         std::cout « "testGridSize: Passed" « std::endl;
00354 //     } else {
00355 //         std::cout « "testGridSize: Failed" « std::endl;
00356 //         fails++;
00357 //     }
00358
00359 //     if (testGridSquareCenters()){
00360 //         std::cout « "testGridSquareCenters: Passed" « std::endl;
00361 //     } else {
00362 //         std::cout « "testGridSquareCenters: Failed" « std::endl;
00363 //         fails++;
00364 //     }
00365
00366 //     if (testCurrentRowCol()){
00367 //         std::cout « "testCurrentRowCol: Passed" « std::endl;
00368 //     } else {
00369 //         std::cout « "testCurrentRowCol: Failed" « std::endl;
00370 //         fails++;
00371 //     }
00372
00373 //     if (testCurrentSquare()){
00374 //         std::cout « "testCurrentSquare: Passed" « std::endl;
00375 //     } else {
00376 //         std::cout « "testCurrentSquare: Failed" « std::endl;
00377 //         fails++;
00378 //     }
00379
00380 //     if (testGetSquareByPos()){
00381 //         std::cout « "testGetSquareByPos: Passed" « std::endl;
00382 //     } else {
```

```
00383 //         std::cout « "testGetSquareByPos: Failed" « std::endl;
00384 //         fails++;
00385 //     }
00386
00387 //     if (testNextRoad()){
00388 //         std::cout « "testNextRoad: Passed" « std::endl;
00389 //     } else {
00390 //         std::cout « "testNextRoad: Failed" « std::endl;
00391 //         fails++;
00392 //     }
00393
00394 //     if (testRead()){
00395 //         std::cout « "testRead: Passed" « std::endl;
00396 //     } else {
00397 //         std::cout « "testRead: Failed" « std::endl;
00398 //         fails++;
00399 //     }
00400
00401 //     if (testWrite()){
00402 //         std::cout « "testWrite: Passed" « std::endl;
00403 //     } else {
00404 //         std::cout « "testWrite: Failed" « std::endl;
00405 //         fails++;
00406 //     }
00407
00408 //     if (testObjectList()){
00409 //         std::cout « "testObjectList: Passed" « std::endl;
00410 //     } else {
00411 //         std::cout « "testObjectList: Failed" « std::endl;
00412 //         fails++;
00413 //     }
00414
00415 //     std::cout « "Making random map:" « std::endl;
00416 //     if (testRandomMap()){
00417 //         std::cout « "testRandom: Passed" « std::endl;
00418 //     } else {
00419 //         std::cout « "testRandom: Failed" « std::endl;
00420 //     }
00421
00422 //     //testRandom2();
00423
00424 //     if (fails == 0){
00425 //         std::cout « "All Level test passed" « std::endl;
00426 //     } else {
00427 //         std::cout « fails « " Level test failed" « std::endl;
00428 //     }
00429
00430 //     return fails;
00431 // }
```

7.18 ObjectTests.cpp

```
00001 // #include "object.hpp"
00002 // #include "vector2d.hpp"
00003 // #include "attack_types.hpp"
00004 // #include "level.hpp"
00005 // #include <iostream>
00006
00007 // bool testObjectHealth() {
00008 //     Vector2D position(0, 0);
00009 //     Level level(1000, 100, 3);
00010 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00011 //     return obj.get_health() == 100;
00012 // }
00013
00014 // bool testObjectPosition() {
00015 //     Vector2D position(0, 0);
00016 //     Vector2D newPosition(10, 20);
00017 //     Level level(1000, 100, 3);
00018 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00019
00020 //     // Check the initial position
00021 //     if (obj.get_position() != position) {
00022 //         return false;
00023 //     }
00024
00025 //     // Set a new position and check if it's updated
00026 //     obj.set_position(newPosition);
00027 //     if (obj.get_position() != newPosition) {
00028 //         return false;
00029 //     }
00030
00031 //     return true;
```

```
00032 // }
00033
00034 // bool testObjectDamage() {
00035 //     Vector2D position(0, 0);
00036 //     Level level(1000, 100, 3);
00037 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00038 //     return obj.get_damage() == 20;
00039 // }
00040
00041 // bool testObjectRange() {
00042 //     Vector2D position(0, 0);
00043 //     Level level(1000, 100, 3);
00044 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00045 //     return obj.get_range() == 10;
00046 // }
00047
00048 // bool testObjectAttackSpeed() {
00049 //     Vector2D position(0, 0);
00050 //     Level level(1000, 100, 3);
00051 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00052 //     return obj.get_attack_speed() == 1000;
00053 // }
00054
00055 // bool testObjectGetType() {
00056 //     Vector2D position(0, 0);
00057 //     Level level(1000, 100, 3);
00058 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00059 //     return obj.get_type() == BASIC;
00060 // }
00061
00062 // bool testObjectGainDamage() {
00063 //     Vector2D position(0, 0);
00064 //     Level level(1000, 100, 3);
00065 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00066 //     obj.gain_damage(5);
00067 //     return obj.get_damage() == 25;
00068 // }
00069
00070 // bool testObjectGainHealth() {
00071 //     Vector2D position(0, 0);
00072 //     Level level(1000, 100, 3);
00073 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00074 //     obj.gain_health(10);
00075 //     return obj.get_health() == 110;
00076 // }
00077
00078 // bool testObjectGainRange() {
00079 //     Vector2D position(0, 0);
00080 //     Level level(1000, 100, 3);
00081 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00082 //     obj.gain_range(2);
00083 //     return obj.get_range() == 12;
00084 // }
00085
00086 // bool testObjectGainAttackSpeed() {
00087 //     Vector2D position(0, 0);
00088 //     Level level(1000, 100, 3);
00089 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00090 //     obj.gain_attack_speed(200);
00091 //     return obj.get_attack_speed() == 1200;
00092 // }
00093
00094 // bool testObjectDistanceTo() {
00095 //     Vector2D position(0, 0);
00096 //     Level level(1000, 100, 3);
00097 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00098 //     Vector2D targetPosition(3, 4);
00099 //     double distance = obj.distance_to(targetPosition);
00100 //     // Check if the distance is calculated correctly (considering the distance formula)
00101 //     return distance == 5.0;
00102 // }
00103
00104 // bool testObjectLoseHealth() {
00105 //     Vector2D position(0, 0);
00106 //     Level level(1000, 100, 3);
00107 //     Object obj(level, 100, 20, 10, 1000, position, BASIC);
00108 //     obj.lose_health(30);
00109 //     return obj.get_health() == 70;
00110 // }
00111
00112 // static int object_tests() {
00113 //     int testsFailed = 0;
00114
00115 //     if (testObjectHealth()) {
00116 //         std::cout « "testObjectHealth: Passed" « std::endl;
00117 //     } else {
00118 //         std::cout « "testObjectHealth: Failed" « std::endl;
```

```
00119 //         testsFailed++;
00120 //     }
00121
00122 //     if (testObjectPosition()) {
00123 //         std::cout « "testObjectPosition: Passed" « std::endl;
00124 //     } else {
00125 //         std::cout « "testObjectPosition: Failed" « std::endl;
00126 //         testsFailed++;
00127 //     }
00128
00129 //     if (testObjectDamage()) {
00130 //         std::cout « "testObjectDamage: Passed" « std::endl;
00131 //     } else {
00132 //         std::cout « "testObjectDamage: Failed" « std::endl;
00133 //         testsFailed++;
00134 //     }
00135
00136 //     if (testObjectRange()) {
00137 //         std::cout « "testObjectRange: Passed" « std::endl;
00138 //     } else {
00139 //         std::cout « "testObjectRange: Failed" « std::endl;
00140 //         testsFailed++;
00141 //     }
00142
00143 //     if (testObjectAttackSpeed()) {
00144 //         std::cout « "testObjectAttackSpeed: Passed" « std::endl;
00145 //     } else {
00146 //         std::cout « "testObjectAttackSpeed: Failed" « std::endl;
00147 //         testsFailed++;
00148 //     }
00149
00150 //     if (testObjectGetType()) {
00151 //         std::cout « "testObjectGetType: Passed" « std::endl;
00152 //     } else {
00153 //         std::cout « "testObjectGetType: Failed" « std::endl;
00154 //         testsFailed++;
00155 //     }
00156
00157 //     if (testObjectGainDamage()) {
00158 //         std::cout « "testObjectGainDamage: Passed" « std::endl;
00159 //     } else {
00160 //         std::cout « "testObjectGainDamage: Failed" « std::endl;
00161 //         testsFailed++;
00162 //     }
00163
00164 //     if (testObjectGainHealth()) {
00165 //         std::cout « "testObjectGainHealth: Passed" « std::endl;
00166 //     } else {
00167 //         std::cout « "testObjectGainHealth: Failed" « std::endl;
00168 //         testsFailed++;
00169 //     }
00170
00171 //     if (testObjectGainRange()) {
00172 //         std::cout « "testObjectGainRange: Passed" « std::endl;
00173 //     } else {
00174 //         std::cout « "testObjectGainRange: Failed" « std::endl;
00175 //         testsFailed++;
00176 //     }
00177
00178 //     if (testObjectGainAttackSpeed()) {
00179 //         std::cout « "testObjectGainAttackSpeed: Passed" « std::endl;
00180 //     } else {
00181 //         std::cout « "testObjectGainAttackSpeed: Failed" « std::endl;
00182 //         testsFailed++;
00183 //     }
00184
00185 //     if (testObjectDistanceTo()) {
00186 //         std::cout « "testObjectDistanceTo: Passed" « std::endl;
00187 //     } else {
00188 //         std::cout « "testObjectDistanceTo: Failed" « std::endl;
00189 //         testsFailed++;
00190 //     }
00191
00192 //     if (testObjectLoseHealth()) {
00193 //         std::cout « "testObjectLoseHealth: Passed" « std::endl;
00194 //     } else {
00195 //         std::cout « "testObjectLoseHealth: Failed" « std::endl;
00196 //         testsFailed++;
00197 //     }
00198
00199 //     if (testsFailed == 0) {
00200 //         std::cout « "All tests passed." « std::endl;
00201 //     } else {
00202 //         std::cout « "Some tests failed." « std::endl;
00203 //     }
00204
00205 //     return testsFailed;
```

```
00206 // }
```

7.19 SquareTests.cpp

```
00001 // #include "square.hpp"
00002 // #include <iostream>
00003
00004 // bool testCenter(){
00005 //     Vector2D cent(2, 3);
00006 //     Square sq(cent);
00007 //     return sq.get_center() == cent;
00008 // }
00009
00010 // bool testOccupied(){
00011 //     Vector2D cent(2, 3), cent2(4, 5), cent3(6, 7);
00012 //     Square sq(cent), sq2(cent2), sq3(cent3);
00013 //     sq.occupy_by_grass();
00014 //     sq2.occupy_by_road();
00015 //     sq3.occupy_by_tower();
00016 //     return sq.get_occupied() == grass && sq2.get_occupied() == road && sq3.get_occupied() == tower;
00017 // }
00018
00019 // static int square_test() {
00020 //     int fails = 0;
00021
00022 //     if (testCenter()){
00023 //         std::cout « "testCenter: Passed" « std::endl;
00024 //     } else {
00025 //         std::cout « "testCenter: Failed" « std::endl;
00026 //         fails++;
00027 //     }
00028
00029 //     if (testOccupied()){
00030 //         std::cout « "testOccupied: Passed" « std::endl;
00031 //     } else {
00032 //         std::cout « "testOccupied: Failed" « std::endl;
00033 //         fails++;
00034 //     }
00035
00036 //     if (fails == 0){
00037 //         std::cout « "All Square test passed" « std::endl;
00038 //     } else {
00039 //         std::cout « fails « " Square test failed" « std::endl;
00040 //     }
00041
00042 //     return fails;
00043 // }
```