Computer Science Project 3 Report

# Description of public member functions

## StudentWorld Class

* Player\* getPlayer();

This function returns a pointer to the player object. Other objects in the game may need to access information regarding the player or change some attribute of the player. For example, the restore-health object needs to restore the player’s hit points and the ammo object needs to increase the player’s ammo count, when the player lands on any of these goodies.

* bool canPlayerMoveThere(int x, int y, int moveDirection);

This function is called when a player is prompted to move in a particular direction. The function ensures that the player isn’t trying to move onto an obstacle (walls, holes, robots, robot factories and immovable boulders). If the object in the player’s way is a boulder, another function canBouldersMoveThere is called to find out if the boulder is immovable or not. This function is implemented using a loop that checks the ID of all items on the particular coordinate the player is trying to move to. If the ID of any obstacle is found, the function immediately returns false. Otherwise, if no obstacles are found, the function returns true.

* bool canBouldersMoveThere(int x, int y);

The function is called to find out if a boulder can be moved to a particular slot. Every object in the game except holes (and bullets) are obstacles to the boulder. This function is required to find out whether a boulder is immovable or not. Similar to the function above, this function is implemented using a loop that checks the ID of all items on the particular coordinate the boulder is being pushed to. If the ID of any obstacle is found, the function immediately returns false. Otherwise, if no obstacles are found, the function returns true.

* bool isBoulderOnHole(int x, int y);

The ‘hole’ object calls this function to check if a boulder is currently on its position. Hence, the function searches through the vector, consisting of pointers to actors, to check if any boulder object shares the same coordinates as the hole. A loop is implemented to go through all the actors. The function immediately returns true if a boulder is found on the hole; the function returns false if no such boulder object is found.

* bool isPlayerOnMe(int x, int y);

The function returns true if the player’s position matches the position specified by the parameters (and false otherwise). The goodie objects such as ammo and extra life use this function to determine whether the player has obtained the goodie or not.

* void createBulletAt(int x, int y, int direction);

This function creates a bullet at a particular set of coordinates, and assigns the bullet a specific direction. The bullet object is dynamically allocated and the pointer to the object is stored in the vector STL. This function is called by the player and robot objects when needed.

* bool canBulletDamage(int x, int y);

Checks if the bullet is currently on an object which it can damage, such as player, robots and boulders. The function first finds actors that share the same position as the bullet (passed in as the parameters). If any of these actors are player, robots or boulders, the object is prompted to take damage and the function returns true. This return value will be used to determine whether the bullet should be erased or not.

* bool canBulletDie(int x, int y);

The bullet could also ‘die’ without causing any damage, i.e. by colliding with a wall or robot factory. Similar to the function above, this function first finds actors that share the same position as the bullet (passed in as the parameters). If any of these actors are walls or robot factories, the function returns true, and the bullet object will use this return value to set its status as ‘not alive’.

* bool isPlayerInMyLOS(int x, int y, int direction);

This function determines if the player object is in the line of sight of a robot- that can shoot-positioned at the coordinates specified by the function’s parameters. Using the robot’s position as a starting point, the function determines in which direction to start searching by considering the direction the robot faces (which is also passed in as a parameter). From the starting point, the function searches every position (in a particular direction) for the player or an obstruction, such as a boulder, wall, robot or robot factory. If an obstruction is found the function immediately returns false, and if a player is found the function returns true.

* bool canRobotMoveThere(int x, int y);

The function determines if a boulder can be moved to a particular slot. Walls, boulders, holes, other robots and robot factories are obstacles to a robot. This function is called before a robot is prompted to move to a particular slot. This function is implemented using a loop that checks the ID of all items on the particular coordinate the boulder is being pushed to. If the ID of any obstacle is found, the function immediately returns false. Otherwise, if no obstacles are found, the function returns true.

* bool stealGoodieIfPossible(int x, int y, Kleptobot\* ptr);

This function checks if a goodie is present in the same position as a kleptobot (whose coordinates are passed in as parameters). The function checks if an actor in the position of interest is a goodie by dynamically casting the actor’s pointer to a goodie pointer. If this pointer is not a null pointer and the goodie’s ID is not IID\_JEWEL, the kleptobot is prompted to store the ID of this goodie, and the goodie is deleted. The function returns true if the kleptobot is successfully able to steal a goodie and it returns false otherwise.

* void createGoodieAt(int x, int y, int ID);

A kleptobot calls this function when it dies to drop the goodie it stole. The goodie is created at the position the kleptobot dies. A new goodie of a particular type, as specified by the ID parameter, is dynamically allocated and its pointer is stored in the vector STL.

* void produceKleptobotIfPossible(int x, int y, std::string identity);

This function is called by kleptobot factory objects to create kleptobots (if possible) during a tick. If a kleptobot is present on the factory, the function immediately returns. The function then counts how many kleptobots of any kind are present in a 7x7 square region with the robot factory in the center. If less than 3 kleptobots are present in this region, then there is a 1 in 50 chance that the function dynamically allocates a new kleptobot; otherwise the function just returns.

**Class Inheritance Diagram**

## Actor Class

* Actor(int imageID, int startX, int startY, Direction startDirection, StudentWorld\* studentWorld, int hitpoints);

Constructs an actor object by initializing the values of local variables and calling the constructor of the GraphObject base class with relevant parameters.

* virtual void doSomething() = 0;

As an actor object will never be created, I chose to make this function virtual. Moreover most derived class objects do something during a tick, so I declared this function as purely virtual.

* void takeDamage();

This function reduces the hitpoints of the object by two units.

* void restoreHitpoints();

Sets the hitpoints of the actor to 20 units. (Only player ever calls this function)

* void createBullet();

This function gets the direction of the actor who is shooting and then calls the createBulletAt function in the StudentWorld class to create a new bullet with the correct direction.

* int getID() const;

Returns the ID of the actor.

* int getHitpoints() const;

Returns the hitpoints of the actor.

* bool getAlive() const;

Returns whether the actor is live or not.

* void setAlive(bool status);

Sets the status of an actor as either alive or dead (true or false).

* StudentWorld\* getWorld() const;

Returns a pointer to the StudentWorld class. This pointer is used to access the public member function in the StudentWorld class.

## Player Class

* Player(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_PLAYER, Direction startDirection = right, int hitpoints = 20, int ammo = 20);

Constructs a player object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters.

* virtual void doSomething();

Checks if user has pressed a key, and appropriately moves the player or prompts the base class to fire a bullet, reducing the ammo count. The function also sets the player dead if the hitpoints drop below zero.

* int getAmmo();

Returns the value of the ammunition the player holds.

* void increaseAmmo();

Increases the ammo by 20. This function is called when the player picks up the ammo goodie.

## Wall Class

* Wall(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_WALL, Direction startDirection = none, int hitpoints = 1);

Constructs a wall object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters.

* virtual void doSomething();

Walls don’t do anything during a tick.

## Exit Class

* Exit(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_EXIT, Direction startDirection = none, int hitpoints = 1);

Constructs an exit object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters.

* virtual void doSomething();

Checks if player is on the exit, and ensures that no jewels are present. Then takes player to next level.

* void setReveal(bool status);

Changes the visibility status of the exit object.

* bool getReveal();

Returns the visibility status of the exit object.

## Boulder Class

* Boulder(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_BOULDER, Direction startDirection = none, int hitpoints = 10);

Constructs a boulder object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters.

* virtual void doSomething();

Boulders don’t do anything during a tick.

## Hole Class

* Hole(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_HOLE, Direction startDirection = none, int hitpoints = 1);

Constructs a hole object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters.

* virtual void doSomething();

If a Boulder is present on the hole, the hole destroys itself and prompts the boulder to set itself dead.

## Bullet Class

* Bullet(int startX, int startY, StudentWorld\* studentWorld, int startDirection\_no, int imageID = IID\_BULLET, int hitpoints = 1);

Constructs a bullet object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters.

* virtual void doSomething();

If there are no obstructions of any sort the bullet moves one step in its current direction during a tick. If there is an obstruction, the function determines whether to simply destroy the bullet or cause to damage to the obstacle and then destroy the bullet.

## Goodie Class

* Goodie(int startX, int startY, StudentWorld\* studentWorld, int imageID, Direction startDirection, int hitpoints);

Constructs a goodie object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters.

* virtual void doSomething();

Checks if the goodie is still alive. If it’s not, the function just returns.

* void takeGoodieAndPlaySound();

Sets the goodie’s status as dead and plays the sound for the player getting a goodie.

## Jewel Class

* Jewel(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_JEWEL, Direction startDirection = none, int hitpoints = 1);

Constructs a Jewel object by initializing the values of local variables and calling the constructor of the Goodie base class with relevant parameters.

* virtual void doSomething();

Checks if player has grabbed the goodie or not, and implements its effects appropriately.

## Restore Health Class

* Restore\_Health(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_RESTORE\_HEALTH, Direction startDirection = none, int hitpoints = 1);

Constructs a Restore Health object by initializing the values of local variables and calling the constructor of the Goodie base class with relevant parameters.

* virtual void doSomething();

Checks if player has grabbed the goodie or not, and implements its effects appropriately.

## Extra Life Class

* Extra\_Life(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_EXTRA\_LIFE, Direction startDirection = none, int hitpoints = 1);

Constructs an Extra Life object by initializing the values of local variables and calling the constructor of the Goodie base class with relevant parameters.

* virtual void doSomething();

Checks if player has grabbed the goodie or not, and implements its effects appropriately.

## Ammo Class

* Ammo(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_AMMO, Direction startDirection = none, int hitpoints = 1);

Constructs an Ammo object by initializing the values of local variables and calling the constructor of the Goodie base class with relevant parameters.

* virtual void doSomething();

Checks if player has grabbed the goodie or not, and implements its effects appropriately.

## Robot Class

* Robot(int startX, int startY, StudentWorld\* studentWorld, int imageID, Direction startDirection, int hitpoints);

Constructs a robot object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters.

* bool shouldIdoSomething();

Returns whether the robots should do something during the given tick.

## Snarlbot Class

* Snarlbot(int startX, int startY, StudentWorld\* studentWorld, int startDirection\_no, int imageID = IID\_SNARLBOT, int hitpoints = 10);

Constructs a Snarlbot object by initializing the values of local variables and calling the constructor of the Robot base class with relevant parameters

* virtual void doSomething();

Checks if the robot is dead, sets the robot as dead if hitpoints reach zero, moves the robot and fires a bullet if a player is in light of sight.

## Kleptobot Class

* Kleptobot(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_KLEPTOBOT, Direction startDirection = right, int hitpoints = 5);

Constructs a Kleptobot object by initializing the values of local variables and calling the constructor of the Robot base class with relevant parameters

* virtual void doSomething();

Checks if the robot is dead, sets the robot as dead if hitpoints reach zero, moves the robot, steals goodies and fires a bullet if a player is in light of sight and the kleptobot is an angry type.

* void setStolenGoodie(int type);

Sets the stolen goodie member variable to a particular type.

## Angry Kleptobot Class

* Angry\_Kleptobot(int startX, int startY, StudentWorld\* studentWorld, int imageID = IID\_ANGRY\_KLEPTOBOT, Direction startDirection = right, int hitpoints = 8);

Constructs an angry kleptobot object by initializing the values of local variables and calling the constructor of the kleptobot base class with relevant parameters

## Kleptobot Factory Class

* Kleptobot\_Factory(int startX, int startY, StudentWorld\* studentWorld, std::string identity = "normal", int imageID = IID\_ROBOT\_FACTORY, Direction startDirection = none, int hitpoints = 1);

Constructs a kleptobot factory object by initializing the values of local variables and calling the constructor of the Actor base class with relevant parameters

* virtual void doSomething();

Calls the produceKleptobotIfPossible function in the StudentWorld class, thereby resulting a 1 in 50 chance that the factory produces a kleptobot during a tick.

## Angry Kleptobot Factory Class

* Angry\_Kleptobot\_Factory(int startX, int startY, StudentWorld\* studentWorld, std::string identity = "angry");

Constructs an angry kleptobot factory object by initializing the values of local variables and calling the constructor of the kleptobot factory base class with relevant parameters.

# Incomplete Functionalities and Known Bugs

The program satisfies all functionalities listed in the specification and there are no known bugs in the program.

# Design decisions and assumptions

* As it was not specified whether a dropped goodie must appear over or under a jewel, I decided to place the dropped goodie above the jewel (or another goodie). This was done by inserting dropped goodie objects to the beginning of the vector STL.
* As it was not specified whether a robot must appear under or over the goodies, the robots may sometimes move over the goodies, while they may also move under the goodies. (This depends on the position of the object’s pointer in the vector STL).

# Testing methodology

## Actor Class

As this class is an abstract base class, its functionality is tested based on the functioning of its derived classes.

## Player Class

The player class was tested by checking if the player appropriately responded to inputs from the keyboard, and interacted with its environment correctly (for example not run through walls). When commands are given though the keyboard, the player appropriately responds to directional movement and also prompts the base class to create a bullet if the space bar was pressed. When the escape key is pressed the player loses a life and the level is restarted.

## Wall Class

The wall object does not do anything during a given tick, so the wall class was tested by ensuring that the walls were visible when the program was run and that the walls were in their correct place as per as the ‘.dat’ file.

## Exit Class

The functionality of this class was tested by ensuring that the exit became visible when the player collected all the jewels. Moreover, the exit also shouldn’t take the player to the next level unless it becomes visible. The exit also appropriately responds when a player lands on it by prompting the program to complete the current level and move on to the next level. The exit class also shouldn’t interact with other objects in the game.

## Boulder Class

The boulder object does not do anything during a tick. The class is tested by ensuring the boulder correctly responds when the player pushes it (by either moving or not moving due to an obstacle). The boulder class must also ensure the boulder is set dead when its hitpoints reach zero.

## Hole Class

During every tick, the hole checks if a boulder is on it. And if a boulder is on the hole, the hole must destroy itself and prompt the particular boulder to also set itself as dead. This functionality was observed to work correctly during gameplay.

## Bullet Class

The bullet class was tested by ensuring that the bullet moved every tick (i.e. it moved smoothly), and that it caused damage or died without causing damage depending on the object it impacted. The bullet also had to be oriented appropriately, and this could be observed in the game as the bullet shared the same direction the shooter had when it shot the bullet. The bullet also shouldn’t execute these actions too fast, as the bullet may not be visible if the shooter is too close to his target. By appropriately placing the pointer to the bullet object in the STL, the bullet can be visually observed as it moves from its shooter to its target.

## Goodie Class

The goodie class is a base class, so its functionality was tested based on the functionality of its derived classes. The goodie, however, does have the responsibility to not allow the dead objects of its derived classes to do anything during a tick, and the goodie indeed does nothing after its set dead (i.e. if it still isn’t removed from the vector STL).

## Jewel Class

The jewels must simply check if a player on it. And if a player is on the jewel, the jewel should set itself dead and increase the player’s score. This functionality could indeed be observed during gameplay.

## Restore Health Class

The restore health class was tested by ensuring that the layers health was restores to 100% when the player picked up this goodie. This trait could be observed during gameplay.

## Extra Life Class

This class was tested by ensuring that the player received an extra life when this goodie was picked up. And this functionality could be observed in the game.

## Ammo Class

The ammo class was tested by checking whether the ammo count increased or not, when the player picked up this goodie. And indeed, the ammo count did increase the player picked it up.

## Robot Class

As the robot class is an abstract base class, its functionality was tested based on its derived classes.

## Snarlbot Class

The snarlbot was tested by checking whether the snarlbot appropriately moved (at the right pace) and correctly responded to the actions of other objects in the game. The snarlbot indeed didn’t move every tick (as it could be observed that the robots moved slower than the bullets). The sandlots also move only in to complementary directions as stated in the specification. The robot also doesn’t move past obstructions and instead changes direction. When a player comes in the snarlbot’s line of sight, the snarlbot fires at the player. Hence, the snarlbot satisfies all its functional specifications.

## Kleptobot Class

The kleptobots were tested by ensuring that they had random motion (as per as the specification), interacted with other game objects appropriately and stole goodies when they could do so. The kleptobot didn’t move pass any obstacles, and instead did change directions. Kleptobots are also successful in stealing goodies (in accordance with its 1 in 10 chance of doing so). When kleptobots were killed they also correctly dropped the goodie at the position they died.

## Angry Kleptobot Class

The angry kleptobot should do everything a normal kleptobot does and also shoot if a player is in its line of sight. These angry kleptobots also have more hitpoints, which could be verified, as the angry ones took more bullets to kill.

## Kleptobot Factory Class

This class was tested by ensuring that the factory produced the kleptobots at the same rate as the one in the original game. The factory also shouldn’t produce any kleptobots if there were at least 3 robots near the factory or if a kleptobot was already present on the factory. These conditions were met as they could be observed during gameplay.

## Angry Kleptobot Factory Class

This factory is exactly similar in behavior to the normal kleptobot factory except that this factory should produce angry kleptobots.