Nilay Modi

404-429-944

# Boulder Bash Report

1) Actor (sublcass of GraphObject)

**Actor(int imageID, int startX, int startY, StudentWorld\* world, Direction dir = none);**

* Constructs Actor object with the correct x and y coordinates, correct world, and correct direction
* Set input ‘world’ to private instance variable ‘m\_world’
* Set’s visible to true

**StudentWorld\* getWorld(){**

**return m\_world;**

**}**

— Returns the StudentWorld object, associated with the Actor

**virtual void setHealth(int health){}**

* Virtual because some subclass actors will have a health to set, while other won’t.
* It’s not a mandatory function as only some actors are alive

**virtual int getHealth(){return 0;}**

* Virtual because some subclass actors will have a health to return, while other won’t.
* It’s not a mandatory function as only some actors are alive, the ones that are not will return 0

**virtual bool blocksPlayer(){**

**return true;**

**}**

* Virtual because some subclass actors will override this method so that it doesn’t block player
* Default actor blocks player

**virtual void gotHurt(){};**

* Virtual because some subclass actors will have be able to get hurt, while others won’t
* It’s not a mandatory function as only some actors are alive and can get hurt. By default, actors don’t get hurt

**virtual void doSomething() = 0;**

* Pure virtual function because every subclass of Actor has to do something and it will be different for every class

**virtual int type() = 0;**

* Pure virtual function because every subclass of Actor has a different type

**NO TESTING BECAUSE IT’S AN ABSTRACT CLASS**

2) Alive (subclass of Actor)

**Alive(int imageID, int startX, int startY, StudentWorld\* world,Direction dir = none);**

* Constructs Alive object with the correct x and y coordinates, correct world, and correct direction

**void setHealth(int health){**

**m\_health = health;**

**}**

- Function that set’s health of Alive objects

**int getHealth(){**

**return m\_health;**

**}**

- Function that return the health of Alive objects

**virtual void doSomething() = 0;**

* Pure virtual function because every subclass of Actor has to do something and it will be different for every class

**virtual int type() = 0;**

* Pure virtual function because every subclass of Actor has a different type

**NO TESTING BECAUSE IT’S AN ABSTRACT CLASS**

3) Player (subclass of Alive)

**Player(int startX, int startY, StudentWorld\* world);**

* Constructs Player object with the correct x and y coordinates, correct world and uses IID\_PLAYER image
* Sets health to 20
* Sets ammo to 20
* Sets direction to right

**virtual void doSomething();**

* If Player is not alive, do nothing and return
* If escape key pressed, set Player to dead
* If space key pressed, and ammo is greater than 0, Player should fire a bullet
  + - Ammo should be decreased
    - Bullet should be added to Actor’s vector in the StudentWorld
    - SOUND\_PLAYER\_FIRE sound should be played
* If direction key pressed
  + - Ask StudentWorld if player can move one step in that direction (implemented in StudentWorlds’s **canPlayerMove()** because the world knows where everything is. This way the Player is independent)
    - If yes, move the player to that position

**virtual void gotHurt();**

* Reduces Player’s health by two
* If Player’s health > 0, play SOUND\_PLAYER\_IMPACT sound
* Otherwise, play SOUND\_PLAYER\_DIE sound

**int getAmmo(){**

**return m\_ammo;**

**}**

- Return ammo of the Player

**void setAmmo(int a){**

**m\_ammo = a;**

**}**

- Set ammo of the player

**virtual int type(){**

**return IID\_PLAYER;**

**}**

* Returns IID\_PLAYER which will be used to determine the type of Actor in StudentWorld

**TESTING**

Compare level data file with where the Player was created to make sure it is placed in the correct location. Make sure that the Player can’t move through actors whose blockPlayer() return true. Also, direction key responds to movement in that direction Then check, if when the escape key is pressed, player loses life. If space bar is pressed bullet is fired in the correct direction, ammo should reduce by 1, ammo can’t be negative. Also check if players move boulders appropriately i.e not go through walls, boulder doesn’t go over boulder etc. Next, I placed one SnarlBot in the game to test if correct sounds play when the player gets shot and health decreases by 2.

4) Wall (subclass of Actor)

**Wall(int startX, int startY, StudentWorld\* world);**

* Constructs Wall object with the correct x and y coordinates, correct world, and no direction and used IID\_WALL image

**virtual void doSomething(){**

**}**

- Wall does nothing each tick

**virtual int type(){**

**return IID\_WALL;**

**}**

* Returns IID\_WALL which will be used to determine the type of Actor in StudentWorld

**TESTING**

Compare level data file with where the Wall was created to make sure it is placed in the correct location. Also, make sure wall does nothing

5) Boulder (subclass of Alive)

**Boulder(int startX, int startY, StudentWorld\* world);**

* Constructs Boulder object with the correct x and y coordinates, correct world, and no direction and uses IID\_BOULDER image
* Set’s health to 10

**virtual void doSomething(){**

**}**

- Boulder does nothing each tick

**virtual void gotHurt(){**

**setHealth(getHealth()-2);**

**}**

- Boulder’s health decreases by 2 every time it gets hurt

**virtual int type(){**

**return IID\_BOULDER;**

**}**

* Returns IID\_BOULDER which will be used to determine the type of Actor in StudentWorld

**EXTRA**

* The boulder getting pushed part was not implemented in the Boulder class but rather in the StudentWorld class under the methods of **canPlayerMove(), canBoulderMove()**. This is because pushing a Boulder is not something a Boulder does, but rather a reaction to the movement of the Player.

**TESTING**

Check if boulder is placed in the correct location. When player shoots a boulder, I checked if the boulder exploded after 5 shots. Boulder’s do nothing. When pushed on a hole, the Boulder should go away. Boulder’s can’t go above anything else.

6) Bullet (subclass of Alive)

**Bullet(int startX, int startY, StudentWorld\* world, Direction dir, Actor\* owner);**

* Constructs Bullet object with the correct x and y coordinates, correct world, and correct direction, correct owner and uses IID\_BULLET image
* Set’s health to 1

**virtual void doSomething();**

* If Bullet is not currently alive, return and do nothing
* Check if bullet can
* Depending on direction of Bullet
  + - Ask StudentWorld if Bullet can move one step in that direction (implemented in StudentWorld’s **canBulletMove()** because the world knows where everything is. This way the Bullet is independent)
    - If yes, move the Bullet to that position
    - Otherwise, kill the Bullet

**virtual bool blocksPlayer(){**

**return false;**

**}**

- Overrides Actor’s default true to false. Since a Player can be hit by a bullet

**virtual int type(){**

**return IID\_BULLET;**

**}**

* Returns IID\_BULLET which will be used to determine the type of Actor in StudentWorld

**EXTRA**

* Bullet damaging Player, robots or boulder is implemented in **canBulletMove()**. This is because the world knows where everything is and what to do when bullet hits a certain Actor (points for killing are given here also). This way the Bullet is independent.

**TESTING**

Check if bullet comes out of the right direction. Make sure that they hurt the object being shot at and not go through bots, and player. Also the bullet checks to make sure that the bullet fired by a player, and killing a bot, increases score. Bullet’s should go through holes and pickup able items. Should not go through factories and walls

7) Hole (subclass of Alive)

**Hole(int startX, int startY, StudentWorld\* world);**

* Constructs Hole object with the correct x and y coordinates, correct world, and no direction and uses IID\_HOLE image
* Set’s health to 1

**virtual void doSomething();**

* If Hole is not currently alive, return and do nothing
* Otherwise, ask StudentWorld to check if Boulder is above Hole (implemented in StudentWorld’s **isBoulderAboveHole()** because the world knows where everything is. This way the Boulder and Hole are independent)
  + - If true, set Hole to dead

**virtual int type(){**

**return IID\_HOLE;**

**}**

* Returns IID\_HOLE which will be used to determine the type of Actor in StudentWorld

**EXTRA**

* Setting the Boulder to dead when Boulder is above hole is implemented in **isBoulderAboveHole()** because the world knows where everything is. This way the Boulder and Hole are independent

**TESTING**

Hole’s do nothing except check if a boulder is place above it. If a boulder is above it, the hole should die.

8) PickupableItem (subclass of Alive)

**PickupableItem (int imageID, int startX, int startY, StudentWorld\* world, int scoreVal);**

* Constructs Pickupable object with the correct image, correct x and y coordinates, correct world, and no direction
* Set health to 1
* Set m\_scoreValue to scoreVal (i.e points gained when picking up said item)

**virtual void doSomething();**

* If PickupableItem is not currently alive, return and do nothing
* Otherwise, ask StudentWorld if Player is above PickupableItem, do it’s respective task (implemented in StudentWorld’s **isPlayerAbovePickupableItem()** because the world knows where everything is. This way the PickupableItem and Player are independent of each other)
  + - Play SOUND\_GOT\_GOODIE sound
    - set PickupableItem to dead

**int getScoreValue(){**

**return m\_scoreValue;**

**}**

- Returns the m\_scoreValue private variable

**virtual bool blocksPlayer(){**

**return false;**

**}**

* Overrides Actor’s default true to false. Since a Player can go above and pick up a PickupableItem

**virtual int type() = 0;**

* Pure virtual function because every subclass of PickupableItem has a different type

**EXTRA**

Increasing score by x amount of points and doing something else for picking up a PickupableItem is done by the StudentWorld’s **isPlayerAbovePickupableItem()**. This way if player is above said a PickupableItem, the function increases the score accordingly and does anything else associated with picking up item in one function based on the item’s type. Now subclasses of PickupableItem don’t have to worry about increasing score, ammo, health, lives etc. It’s all implemented in one place.

**NO TESTING BECAUSE IT’S AN ABSTRACT CLASS**

9) Jewel (subclass of PickupableItem)

**Jewel(int startX, int startY, StudentWorld\* world);**

* Constructs Jewel object with the correct x and y coordinates, correct world, no direction, uses IID\_JEWEL image and m\_scoreValue of 50 (passed to PickupableItem’s constructor)

**virtual int type(){**

**return IID\_JEWEL;**

**}**

* Returns IID\_JEWEL which will be used to determine the type of Actor in StudentWorld

**TESTING**

Make sure that the Jewel does not block player. When player steps over Jewel the item should give 50 points and then set itself to dead and disappear. Make sure that the sound of picking up is played correctly.

10) ExtraLife (subclass of PickupableItem)

**ExtraLife(int startX, int startY, StudentWorld\* world);**

* Constructs ExtraLife object with the correct x and y coordinates, correct world, no direction, uses IID\_EXTRA\_LIFE image and m\_scoreValue of 1000 (passed to PickupableItem’s constructor)

**virtual int type(){**

**return IID\_EXTRA\_LIFE;**

**}**

* Returns IID\_EXTRA\_LIFE which will be used to determine the type of Actor in StudentWorld

**TESTING**

Make sure that the Extra life does not block player. When player steps over Extra life the item should give 1000 points, and increase life by 1 and then set itself to dead and disappear. Make sure that the sound of picking up is played correctly.

11) RestoreHealth (subclass of PickupableItem)

**RestoreHealth(int startX, int startY, StudentWorld\* world);**

* Constructs RestoreHealth object with the correct x and y coordinates, correct world, no direction, uses IID\_RESTORE\_HEALTH image and m\_scoreValue of 500 (passed to PickupableItem’s constructor)

**virtual int type(){**

**return IID\_RESTORE\_HEALTH;**

**}**

* Returns IID\_RESTORE\_HEALTH which will be used to determine the type of Actor in StudentWorld

**TESTING**

Make sure that the Restore health does not block player. When player steps over Restore health the item should give 500 points, and increase player’s health to 20 and then set itself to dead and disappear. Make sure that the sound of picking up is played correctly.

12) Ammo (subclass of PickupableItem)

**Ammo(int startX, int startY, StudentWorld\* world);**

* Constructs Ammo object with the correct x and y coordinates, correct world, no direction, uses IID\_AMMO image and m\_scoreValue of 100 (passed to PickupableItem’s constructor)

**virtual int type(){**

**return IID\_AMMO;**

**}**

* Returns IID\_AMMO which will be used to determine the type of Actor in StudentWorld

**TESTING**

Make sure that the Ammo does not block player. When player steps over Restore health the item should give 100 points, and increase player’s ammo by 20 and then set itself to dead and disappear. Make sure that the sound of picking up is played correctly.

13) Exit (subclass of Actor)

**Exit(int startX, int startY, StudentWorld\* world);**

* Constructs Exit object with the correct x and y coordinates, correct world, and no direction and uses IID\_EXIT image
* set visible to false

**virtual void doSomething();**

* Ask the StudentWorld if all the jewels have been collected (implemented in StudentWorld’s **collectedAllJewels()** because the world keeps track of all jewels that are collected. This way the Exit is independent)
  + - if exit is not visible
      * set exit to visible
      * play SOUND\_REVEAL\_EXIT sound
* If exit is visible
  + - Ask the StudentWorld if player is above exit(implemented in StudentWorld’s **isPlayerAboveExit()** because the world knows where everything is. This way the Exit is independent)
      * play SOUND\_FINISHED\_LEVEL sound

**virtual int type(){**

**return IID\_EXIT;**

**}**

* Returns IID\_EXIT which will be used to determine the type of Actor in StudentWorld

**virtual bool blocksPlayer(){**

**return false;**

**}**

* Overrides Actor’s default true to false. Since a Player can go above an Exit and complete a level

**EXTRA**

The increasing score for completing a level is implemented in **isPlayerAboveExit()** because the world knows/controls the score. The function also notifies StudentWorld that the level has been completed by setting m\_levelComplete to true.

**TESTING**

Make sure that exit is not revealed at the start of the game. Exit is only revealed after all jewels are collected. The sound of an exit appearing should be player once and that time. One the player is above a visible exit, then the level should be completed, level finished sound should be played. 2000 points plus remaining bonus should be allocated to increase the score. No other object on exit should finish the level. Make sure there is no movement problem on invisible exit, especially when player and bot are trying to move to the same location.

14) Bot (subclass of Alive)

**Bot(int imageID, int startX, int startY, StudentWorld\* world, Direction dir);**

* Constructs Bot object with the correct x and y coordinates, correct world, direction, imageID.
* Sets m\_currentTick to 1
* Sets m\_ticks to (28-levelNumber) / 4 (i.e ticks to wait before acting)
* if m\_ticks less than 3, then m\_ticks = 3

**virtual void gotHurt();**

* Decrease health by 2
* If health > 0, play SOUND\_ROBOT\_IMPACT sound
* Otherwise, play SOUND\_ROBOT\_DIE sound

**int getTicks(){**

**return m\_ticks;**

**}**

- Return m\_ticks

**int getCurrentTick(){**

**return m\_currentTick;**

**}**

- Return m\_currentTick

**void setCurrentTick(int n){**

**m\_currentTick = n;**

**}**

- Set currentTick to a certain number

**virtual void doSomething() = 0;**

* Pure virtual function because every subclass of Bot does something different

**virtual int type() = 0;**

* Pure virtual function because every subclass of Bot has a different type

**NO TESTING BECAUSE IT’S AN ABSTRACT CLASS**

15) SnarlBot (subclass of Bot)

**SnarlBot(int startX, int startY, StudentWorld\* world, Direction dir);**

* Constructs SnarlBot object with the correct x and y coordinates, correct world, correct direction (right or down) and IID\_SNARLBOT image
* Set health to 10

**virtual void doSomething();**

* If SnarlBot is not currently alive, return
* If the current tick is equal to the total ticks
  + - Ask StudentWorld if SnarlBot should fire Bullet (implemented in StudentWorld’s **shouldBotFireBullet()** because the world knows where everything is, especially the player and obstacles. This way the Bot is independent)
      * + Create new bullet in SnarlBot’s direction, add it to StudentWorld’s vector and play SOUND\_ENEMY\_FIRE sound
    - Otherwise, Depending on direction of the SnarlBot,
      * + Ask StudentWorld if the SnarlBot can move one step in that direction, move the SnarlBot (implemented in StudentWorld’s **canBotMove()** because the world knows where everything is in relation to itself. This way the Bot is independent)
        + Otherwise, set direction to the opposite direction
    - Reset current tick to 1
* Otherwise increase current tick by one

**virtual int type(){**

**return IID\_SNARLBOT;**

**}**

* Returns IID\_SNARLBOT which will be used to determine the type of Actor in StudentWorld

**EXTRA**

The points given when a SnarlBot is killed is implemented in **canBulletMove().** The sounds on impact and death is handled by the superclass Bot’s **gotHurt()**. The points are not given in gotHurt() because the owner of the Bullet is an important aspect. A bullet fired by a SnarlBot killing a SnarlBot should not give points to the player.

**TESTING**

Check if the SnarlBot are not acting everytick, but on the 7th tick for the first level, by printing ticks in the console. Make sure that horizontal Snarlbot’s start out facing right and vertical Snarlbot’s start out facing down. The SnarlBot fires bullet only when facing the Player and in the same row or column. If there is an obstacle (boulders, walls, factories, holes) to the Snarlbot, the Snarlbot should change direction. When player shoots SnarlBot, it takes 5 shots for the SnarlBot to die, appropriate impact sound / death should be played.

16) KleptoBot (subclass of Bot)

**KleptoBot(int imageID, int startX, int startY, StudentWorld\* world);**

* Constructs KleptoBot object with the correct x and y coordinates, correct world, right direction and correct image (IID\_KLEPTOBOT or IID\_ANGRY\_KLEPTOBOT)
* Set health to 5
* Set walked distance to 0
* Set distance before turning to a random number between 1 and 6.
* Set picked Goodie to 0 (Pickupable items excluding Jewels)

**virtual void doSomething();**

* If Kleptobot is not currently alive, return and do nothing
* If currentTick is equal to total ticks
  + - Ask StudentWorld If KleptoBot is on goodie and doesn’t already have a goodie (implemented in StudentWorld’s **isKleptoBotOnGoodie()** because the world knows where everything is in relation to itself. This way the KleptoBot is independent). GOODIE IS THE SAME THING AS A PICKUPABLE ITEM EXCEPT JEWELS

There is a 1 in 10 chance that the KleptoBot picks up the goodie

The goodie is set to dead

The SOUND\_ROBOT\_MUNCH is played

Otherwise move()

* + - Otherwise move()
    - Reset current tick to 1
* Increment current tick

**virtual void gotHurt();**

* Call superclass Bot’s gotHurt method
* If KleptoBot is dead and KleptoBot holds a goodie
  + - Drop the goodie where the KleptoBot dies, **dropBackGoodie()**

**void dropBackGoodie();**

* If picked goodie is of type IID\_RESTORE\_HEALTH
  + - Construct RestoreHealth object on KleptoBot’s death location
    - Add RestoreHealth object to the Actor’s vector
* If picked goodie is of type IID\_EXTRA\_LIFE
  + - Construct ExtraLife object on KleptoBot’s death location
    - Add ExtraLife object to the Actor’s vector
* If picked goodie is of type IID\_AMMO
  + - Construct Ammo object on KleptoBot’s death location
    - Add Ammo object to the Actor’s vector

**void move();**

* Set found obstruction to false
* If walked distance is not equal to distance before turning
  + - Depending on direction of the KleptoBot
      * + Ask StudentWorld, if the bot can move one step in that direction (implemented in StudentWorld’s **canBotMove()** because the world knows where everything is in relation to itself. This way the Bot is independent)

move Kleptobot one step in that direction

increment walked distance

return and do nothing

* + - * + Otherwise, set found obstruction to true
* If walked distance equals distance before turning or Kleptobot found obstruction
  + - Set walked distance to 0
    - Set distance before turning to a random number between 1 and 6.
    - Pick a random direction (based on number)
    - Ask StudentWorld if bot can move in random direction, move there and increment walked distance (implemented in StudentWorld’s **canBotMove()** because the world knows where everything is in relation to itself. This way the Bot is independent)
      * + Otherwise, generate a new random direction **(generateUniqueRandoms())**, one that KleptoBot has not already looked in and repeat above step
    - If KleptoBot can’t move anywhere set direction to first looked direction

**int generateUniqueRandoms(std::vector<int> randDirNumbers);**

* generates unique randoms based on input vector, the random numbers signify directions.
* If all directions have been looked at return and do nothing

**int getPickedGoodie(){**

**return m\_pickedGoodie;**

**}**

**-** Return picked goodie id

**void setPickedGoodie(int goo){**

**m\_pickedGoodie = goo;**

**}**

- Set picked goodie to an id

**virtual int type(){**

**return IID\_KLEPTOBOT;**

**}**

- Returns IID\_KLEPTOBOT which will be used to determine the type of Actor in StudentWorld

**EXTRA**

The points given when a KleptoBot is killed is implemented in **canBulletMove().** The sounds on impact and death is handled by the superclass Bot’s **gotHurt()**. The points are not given in gotHurt() because the owner of the Bullet is an important aspect. A bullet fired by a SnarlBot killing a KleptoBot should not give points to the Player.

**TESTING**

Check if the KleptoBot are not acting every tick, but on the 7th tick for the first level, by printing ticks in the console. Make sure that KleptoBot’s start out facing in right direction. If there is an obstacle (boulders, walls, factories, holes) to the Kleptobot, the Kleptobot should change direction. Make sure that Kleptobot moves a set amount of distance and then turns in a random direction, this can be seen by printing distanceBeforeWalking and walkedDistance in the console. If Kleptobot can’t move anywhere it should just change direction. This can tested by placing the bot in locations where it can move, example between walls and holes. When a player shoots KleptoBot, it takes 3 shots for the SnarlBot to die, appropriate impact sound / death should be played.

17) AngryKleptoBot (subclass of KleptoBot)

**AngryKleptoBot(int imageID, int startX, int startY, StudentWorld\* world);**

* Constructs AngryKleptoBot object with the correct x and y coordinates, correct world, right direction and correct image (IID\_ANGRY\_KLEPTOBOT)
* set health to 8

**virtual void doSomething();**

* If AngryKleptobot is not currently alive, return and do nothing
* If currentTick is equal to total ticks
  + - Ask StudentWorld if AngryKleptoBot should fire Bullet (implemented in StudentWorld’s **shouldBotFireBullet()** because the world knows where everything is, especially the player and obstacles. This way the Bot is independent)
      * + Create new bullet in AngryKleptoBot’s direction, add it to StudentWorld’s vector and play SOUND\_ENEMY\_FIRE sound
    - Ask StudentWorld If AngryKleptoBot is on goodie and doesn’t already have a goodie (implemented in StudentWorld’s **isKleptoBotOnGoodie()** because the world knows where everything is in relation to itself. This way the KleptoBot is independent). GOODIE IS THE SAME THING AS A PICKUPABLE ITEM EXCEPT JEWELS

There is a 1 in 10 chance that the AngryKleptoBot picks up the goodie

The goodie is set to dead

The SOUND\_ROBOT\_MUNCH is played

Otherwise move()

* + - Otherwise move()
    - Reset current tick to 1
* Increment current tick

**virtual int type(){**

**return IID\_ANGRY\_KLEPTOBOT;**

**}**

* Returns IID\_ANGRY\_KLEPTOBOT which will be used to determine the type of Actor in StudentWorld

**EXTRA**

The points given when a AngryKleptoBot is killed is implemented in **canBulletMove().** The sounds on impact and death is handled by the superclass Bot’s **gotHurt()**. The points are not given in gotHurt() because the owner of the Bullet is an important aspect. A bullet fired by a SnarlBot killing a AngryKleptoBot should not give points to the Player.

**TESTING**

Same testing as above mentioned KleptoBot, but also test shooting. An AngryKleptoBot should shoot the Player if it is facing Player and in the same row or column as player. AngryKleptobot’s also die after 4 shots from the player not 3.

18) KleptoBotFactory (subclass of Actor)

**KleptoBotFactory(int startX, int startY, StudentWorld\* world, int kbotType);**

* Constructs KleptoBotFactory object with the correct x and y coordinates, correct world, no direction and IID\_ROBOT\_FACTORY image
* Sets Kelptobot to produce (kbotType) to m\_kbotToProduce

**virtual void doSomething();**

* Asks StudentWorld to look 3 spaces up, down, left, right within the constraints of the 15x15 grid for KleptoBots (this is implemented in StudentWorld’s isKleptobotOnLocation() because the world is aware where everything else is in relation to the factory. This way the factory is independent)
* If Kleptobots surrounding the area are less than 3 and there is no Kleptobot on the factory
  + - There is a 1 in 50 chance of the factory spawning a a KleptoBot or AngryKleptoBot (i.e depending on what the factory produces)
    - This produced bot is then added to Actor’s vector
    - SOUND\_ROBOT\_BORN is played

**virtual int type(){**

**return IID\_ROBOT\_FACTORY;**

**}**

* Returns IID\_ROBOT\_FACTORY which will be used to determine the type of Actor in StudentWorld

**TESTING**

Make sure that the correct KleptoBot is produced by the factory depending on what type of factory it is. Test to see that the factory doesn’t produce the respective Kleptobot when there that greater than or equal to 3 Kleptobot’s around the factory and when doesn’t produce when there is a Kleptobot on the factory. Test to see that there is only a 1/50 chance every tick by printing to the console. When robot is born, robot born sound is player. Player should not be able to run through the Factory.

19) StudentWorld (subclass of GameWorld)

**virtual int init();**

* set level complete to false
* if level is greater than 99, return player won
* Based on level
  + - If bad format, error
    - If no more levels player won
* Place all Actors in their respective x,y coordinates based on the Level data file
  + - Add Actor to the vector

TESTING

Mess with the data files and make sure that there is a level error message displayed. When all levels are complete, game finished level is displayed. Make sure that all actors are initialized and placed in the vector.

**virtual int move();**

* Update display text
* Ask the Player to do something
* Go through all the Actors in actor’s vector
  + - Ask each to do something
    - If player is dead, decrease lives and return player died
    - If level complete, return finished level
* Remove dead actors
* Decrease bonus (can’t be negative)
* If player is dead, decrease lives and return player died
* If level complete, return finished level
* Return continue game

TESTING

Check to see if the display text is being updated every tick, especially when some event happens. When an actor has been killed that actor should be removed from the world and deleted appropriately, there should be non memory leak. Bonus should continuously decrease by 1 eery tick, but must not go below zero. If player is dead while playing, the game should display appropriate message. If level is complete, appropriate message should be displayed.

**virtual void cleanUp();**

* Delete Player
* Go through all the actors in vector
  + - Delete actor
    - Erase that part of actor vector

TESTING

Every time game finishes, make sure that there are no extra vector spaces and the objects within it are destroyed. Tested by printing to the console.

**virtual ~StudentWorld();**

* Delete Player
* Go through all the actors in vector
  + - Delete actor
    - Erase that part of actor vector

TESTING

Every time program ends, make sure that there are no extra vector spaces and the objects within it are destroyed. Tested by printing to the console.

The functions below are already tested through the Actor classes, because they are being called from there.

**void addActorToVector(Actor\* a);**

* Put passed actor in the back of actor’s vector

**bool isKleptoBotOnLocation(int x, int y);**

* Go through all Actors in vector
  + - If actor is of type IID\_KLEPTOBOT or IID\_ANGRY\_KLEPTOBOT
      * If actor’s location is same as arguments provided
        + Return True

- Otherwise, return false

**Actor\* isKleptoBotOnGoodie(KleptoBot\* kbot);**

* Go through all Actors in vector
  + - If actor is of type IID\_EXTRA\_LIFE or IID\_RESTORE\_HEALTH or IID\_AMMO
      * If actor’s location is same as arguments provided
        + Return the actor

- Otherwise, return null pointer

**bool shouldBotFireBullet(Bot\* bot);**

‘obstacles’ in this function refers to Actors of type IID\_BOULDER, IID\_WALL, IID\_SNARLBOT, IID\_KLEPTOBOT, IID\_ANGRY\_KLEPTOBOT, IID\_ROBOT\_FACTORY

* If player is in same row as bot
  + - If bot is facing left and the player is in front of bot
      * return false if there are no obstacles
      * Otherwise return true
    - If bot is facing right and the player is in front of bot
      * return false if there are no obstacles
      * Otherwise return true
* If player is in same column as bot
  + - If bot is facing up and the player is in front of bot
      * return false if there are no obstacles
      * Otherwise return true
    - If bot is facing down and the player is in front of bot
      * return false if there are no obstacles
      * Otherwise return true

return false otherwise

**bool canBotMove(int x, int y);**

‘obstacle’ in this function refers to Actors of type IID\_BOULDER , IID\_WALL, IID\_HOLE, IID\_SNARLBOT, IID\_KLEPTOBOT, IID\_KLEPTOBOT , IID\_ROBOT\_FACTORY

* If Player is located on the same coordinates provided
  + - return false

- Go through all Actors in vector

* + - If actor’s location is same as arguments provided
      * If actor is an obstacle, return false
* Otherwise, return true

**bool isPlayerAboveExit(Exit\* e);**

* If exit is visible and player is on exit
  + - increase score by current bonus + 2000
    - set level complete to true
    - return true
* Otherwise, return false

**bool collectedAllJewels();**

* Return true if collected jewels equals total jewels
* Otherwise, return false

**bool isPlayerAbovePickupableItem (PickupableItem\* p);**

* If player’s location is same as Pickupable item’s location
  + - Increase score by Pickupable item’s score value
    - If Pickupable item’s type is IID\_JEWEL
      * + increment collected jewels
    - If Pickupable item’s type is IID\_EXTRA\_LIFE
      * + increment lives
    - If Pickupable item’s type is IID\_AMMO
      * + increase ammo by 20
    - return true
* Otherwise, return false

**bool canBulletMove(int x, int y, Actor\* owner);**

* If owner’s type is not IID\_PLAYER
  + - If player is on the same location as arguments provided
      * + call Player’s got hurt function
        + return false
* set returnValue to true
* Go through all the actors in the vector
  + - If actor’s location is on the same location as the arguments provided
      * + If actor’s type is IID\_WALL

set returnValue to false

* + - * + If actor’s type is IID\_ROBOT\_FACTORY

set returnValue to false

* + - * + If actor’s type is IID\_BOULDER

call actor’s got hurt function

set returnValue to false

* + - * + If owner’s type is not IID\_SNARLBOT and the actor’s type is IID\_SNARLBOT

call actor’s got hurt function

if actor is dead and owner’s type is player, increase score by 100

set returnValue to false

* + - * + If owner’s type is not IID\_KLEPTOBOT and the actor’s type is IID\_KLEPTOBOT

call actor’s got hurt function

if actor is dead and owner’s type is player, increase score by 10

set returnValue to false

* + - * + If owner’s type is not IID\_ANGRY\_KLEPTOBOT and the actor’s type is IID\_ANGRY\_KLEPTOBOT

call actor’s got hurt function

if actor is dead and owner’s type is player, increase score by 20

set returnValue to false

* + return returnValue

**bool isBoulderAboveHole(Hole\* h);**

* Go through all the actors
  + - If actor’s type is IID\_BOULDER and location of actor is the same as the hole
      * + kill boulder
        + return true
* Otherwise, return false

**bool canBoulderMove(int x, int y);**

* If actor is on location as the arguments passed
  + - If actor is a hole, return true
    - Otherwise, return false

- Otherwise, return true

**bool canPlayerMove(int x, int y);**

* set blocks player to false

- Go through all the actors

* + - If actor’s location of is the same as arguments provided
      * + If actor’s type is IID\_BOULDER

Depending on direction of the player

If the boulder can move in that direction

Move the boulder one step in that location

* + - If actor’s location of is the same as arguments provided
      * + If actor blocks player, set block player to true
* If blockPlayer is true, return false
* Otherwise, return true