Ashley Chang

504449890

Project 3 Report

**Descriptions of Classes and Public Member Functions**

**Actors**

1. Actor Class (public GraphObject)

* Actor
  + Constructor for the Actor class. It takes in a pointer to the student world, an ID that is used to identify the object, an x and y coordinate to mark its starting position, a starting direction, and the sublevel in which it is located. This is defined here because each actor is inherited from the GraphObject class, and must provide the information required to construct a GraphObject. This is not pure virtual because all actors are constructed in the same manner and constructors cannot be virtual.
* doSomething
  + This function is a pure virtual function. A definition is not provided here because each object in GoonieBlast is inherited from Actor, and each object needs to “do Something,” but have different behaviors. Also, it is a pure virtual function because the Actor class should be an abstract class.
* getWorld
  + This function accesses the private member sw\_ptr and returns a pointer to its contents. This function is defined here because the Actor class is a base class for all of the other classes, and its inherited classes share this functionality. It is not a pure virtual function because each inherited object uses it in the same way.
* isAlive
  + This function accesses the private member m\_alive and returns a Boolean based on its value. This function is defined here because the Actor class is a base class for all of the other classes, and its inherited classes share this functionality. It is not a pure virtual function because each inherited object sues it in the same way.
* setDead
  + This function changes the private member m\_alive to equal false. This function is defined here because the Actor class is a base class for all of the other classes, and its inherited classes share this functionality. It is not a pure virtual function because each inherited object uses it in the same way.
* receiveDamage
  + This function is not defined here because it is a virtual function that only bots use. It is still present in this class, however, because the manner in which it is used requires access to the Actor class (through a pointer). It is not a pure virtual function because not all inherited classes require its functionality and so need not define it.
* ~Actor
  + Destructor for the class, which destructs itself. It is defined here because since it is a virtual destructor of a base class, it needs to call the destructors of its inherited classes before destructing itself. It is pure virtual because the class is abstract, but still must access its derived classes.

2. Moveable Class (public Actor)

* Moveable
  + Constructor for the Moveable class. It takes in a pointer to the student world, an ID used to identify the object, an x and y coordinate to mark its starting position, a starting direction and a sublevel. This is defined here because as a derived class of Actor, it requires certain information in order to be constructed, and must get this information from any derived class that is being instantiated. It is not pure virtual because Moveable objects are created in the same manner and because constructors cannot be virtual.
* receiveDamage
  + This function is not defined here because it is a virtual function that only bots use. It is still present in this class, however, because the manner in which it is used requires access to the Moveable class (through a pointer). It is not a pure virtual function because not all inherited classes require its functionality and so need not define it.
* doSomething
  + This function is a pure virtual function. A definition is not provided here because each derived class of Moveable needs to “do Something” but with different behaviors. Also, it is a pure virtual function because the Moveable class should be an abstract class.
* tryToMove
  + This function takes in an x coordinate and a y coordinate. It then retrieves a vector of actors that are located on that square, and checks each actor to see whether it is a wall, snarlbot, robot factory, or kleptobot. If it is one of these actors, the function returns false; else it returns true. This function is defined here because only objects of or derived from the Moveable class can use its functionality, and do so in the same way (common behavior). It is not pure virtual because it is not used in different ways.
* go
  + This function takes in a direction and sets the actor to that direction. It then checks the next position in that direction and, using the tryToMove function, checks to see whether the player can move to that square. If it can, it does so and returns true; else it remains in the same position and returns false. This function is defined here because only objects of or derived from the Moveable class use its functionality, and do so in the same way (common behavior). It is not pure virtual because it is not used in different ways.
* moveLine
  + This function is not defined here because it is a virtual function that only bots use. It is still present in this class, however, because the manner in which it is used requires access to the Moveable class (through a pointer). It is not a pure virtual function because not all inherited classes require its functionality and so need not define it.
* fire
  + This function gets the x and y coordinates of the Moveable object it is a part of, then, depending on its direction, will allocate a new bullet the spot in front of it. The function then tells the studentWorld to push the bullet into its private m\_actor vector. This function is defined here because only objects of or derived form the Moveable class use its functionality, and do so in the same way (common behavior). It is not pure virtual because it is not used in different ways, and also because not all derived objects use it.
* ~Moveable
  + Destructor for the class, which destructs itself. It is defined here because since it is a virtual destructor of a base class, it needs to call the destructors of its inherited classes before destructing itself. It is pure virtual because the class is abstract, but still must access its derived classes.

3. Stationary Class (public Actor)

* Stationary
  + Constructor for the Stationary class. It takes in a pointer to the student world, an ID used to identify the object, an x and y coordinate to mark its starting position, a starting direction and a sublevel. This is defined here because as a derived class of Actor, it requires certain information in order to be constructed, and must get this information from any derived class that is being instantiated. It is not pure virtual because Stationary objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function is a pure virtual function. A definition is not provided here because each derived class of Stationary needs to “do Something” but with different behaviors. Also, it is a pure virtual function because the Stationary class should be an abstract class.
* ~Stationary
  + Destructor for the class, which destructs itself. It is defined here because since it is a virtual destructor of a base class, it needs to call the destructors of its inherited classes before destructing itself. It is pure virtual because the class is abstract, but still must access its derived classes.

4. Goodie Class (public Stationary)

* Goodie
  + Constructor for the Goodie class. It takes in a pointer to the student world, an ID used to identify the object, an x and y coordinate to mark its starting position, a starting direction and a sublevel. This is defined here because as a derived class of Stationary, it requires certain information in order to be constructed, and must get this information from any derived class that is being instantiated. It is not pure virtual because Goodie objects are created in the same manner and because constructors cannot be virtual.
* Pickup
  + This function takes in an ID number, and integer representing points. It first checks whether or not a player is on the same spot as the goodie object. If it is, then the function sets the goodie object to dead, plays the sound that indicates goodie pickup, and adjusts points/ammo/health accordingly. This function is defined here because all Goodie objects use its functionality, and do so in the same way. It is not pure virtual because it is not used in different ways by derived classes.
* doSomething
  + This function is a pure virtual function. A definition is not provided here because each derived class of Goodie needs to “do Something” but with different behaviors. Also, it is a pure virtual function because the Goodie class should be an abstract class.
* ~Goodie
  + Destructor for the class, which destructs itself. It is defined here because since it is a virtual destructor of a base class, it needs to call the destructors of its inherited classes before destructing itself. It is pure virtual because the class is abstract, but still must access its derived classes.

5. Gate Class (public Stationary)

* Gate
  + Constructor for the Gate class. It takes in a pointer to the student world, an ID used to identify the object, an x and y coordinate to mark its starting position, and a sublevel. It initializes its starting direction to none. This is defined here because as a derived class of Stationary, it requires certain information in order to be constructed. It is not pure virtual because Gate objects are created in the same manner and because constructors cannot be virtual; it also does not have any derived classes.
* doSomething
  + This function checks to see whether there is a player on the same square as the Gate object itself. If there is, it sets itself to dead and tells the StudentWorld to change the current sublevel to the one indicated by the gate’s private member m\_gateNumber. This is defined here in this manner because it is the only class that behaves this way. It is not pure virtual because it does not have any derived classes, and its functionality is unique.

6. Wall Class (public Stationary)

* Wall
  + Constructor for the Wall class. It takes in a pointer to the student world, an ID used to identify the object, an x and y coordinate to mark its starting position, and a sublevel. It initializes the starting direction to none. This is defined here because as a derived class of Stationary, it requires certain information in order to be constructed. It is not pure virtual because Wall objects are created in the same manner and because constructors cannot be virtual; it also does not have any derived classes.
* doSomething
  + This function does not do anything. This is defined here in this manner because it is the only class that behaves this way. It is not pure virtual because it does not have any derived classes, and its functionality is unique.

7. KleptoFactory Class (public Stationary)

* KleptoFactory
  + Constructor for the KleptoFactory class. It takes in a pointer to the student world, an ID used to identify the object, an x and y coordinate to mark its starting position, and a sublevel. It initializes the starting direction to none. This is defined here because as a derived class of Stationary, it requires certain information in order to be constructed. It is not pure virtual because KleptoFactory objects are created in the same manner and because constructors cannot be virtual; it also does not have any derived classes.
* doSomething
  + This function finds out whether a kleptoBot standing on the factory. If there is not, it checks the surrounding 3x3 area again for kleptoBots. If there are less than 3 kleptoBots in its vicinity, it produces a random number from 0-49. If that number is 0, it allocates a new KleptoBot on its own square, then notifies the studentWorld to push it into its m\_actors vector. This is defined here in this manner because it is the only class that behaves this way. It is not pure virtual because it does not have any derived classes, and its functionality is unique.
* countBots
  + This function checks the 3x3 area around the object. Using a for loop nested inside a for loop, the function gets a vector of actors at each square, then checks the vector for any kleptoBots. If there are any kleptoBots, they are added to a counter, which is returned at the end. This is defined here in this manner because only kleptoBot Factories need this functionality, and it is not shared with any other classes. It is not pure virtual because the KleptoFactory class does not have any derived classes and its functionality is unique.

8. Exit Class (public Goodie)

* Exit
  + Constructor for the Exit class. It takes in a pointer to the student world, an ID used to identify the object, an x and y coordinate to mark its starting position, and a sublevel. It starts with no direction. This is defined here because as a derived class of Goodie, it requires certain information in order to be constructed. It is not pure virtual because Exit objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function uses its pointer to studentWorld to check whether or not all of the hostages and jewels are collected. If so, it checks to see if the Exit object itself is visible, and if not, sets it to visible. It then checks to see if any player is standing on its square, and if so it sets itself to dead by getting “picked up”. This function is defined here because it this class is the only one that behaves in this manner. It is not pure virtual because it does not have any derived classes, and its functionality is unique.

9. Player Class (public Moveable)

* Player
  + Constructor for the Player class. It takes in a pointer to the student world, an x and y coordinate to mark its starting position, and a sublevel. It starts facing right with an ID of IID\_PLAYER. This is defined here because as a derived class of Moveable, it requires certain information in order to be constructed. It is not pure virtual because Player objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function first checks whether the player is dead or has less than 0 health points, in which case it sets itself to dead and returns. If it is alive, it gets input from the user that corresponds to a direction. A switch statement is used, and based on the input, attempts to move one square in that direction. If the escape key is pressed, the player is set to dead, and if the space key is pressed and the player has ammo, the player fires a bullet and subtracts 1 from ammo. This function is defined here because only the Player behaves in this manner and its functionality is not shared with other classes. It is not pure virtual because there are no derived classes that will use it.

10. Hostage Class (public Goodie)

* Hostage
  + Constructor for the Hostage class. It takes in a pointer to the student world, an x and y coordinate to mark its starting position, and a sublevel. It starts with no direction and an ID of IID\_HOSTAGE. This is defined here because as a derived class of Goodie, it requires certain information in order to be constructed. It is not pure virtual because Hostage objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function calls the pickUp function from Moveable with the inputs IID\_HOSTAGE and 0 for points. This call will set the object to dead if a player is on the same square and decrease the hostage count in studentWorld. This function is defined here because hostages are a unique class derived from Goodie, and picking it up has effects that are different from other goodies. It is not pure virtual because it has no derived classes.

11. Jewel Class (public Goodie)

* Jewel
  + Constructor for the Jewel class. It takes in a pointer to the student world, an x and y coordinate to mark its starting position, and a sublevel. It starts with no direction and an ID of IID\_JEWEL. This is defined here because as a derived class of Goodie, it requires certain information in order to be constructed. It is not pure virtual because Jewel objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function calls the pickUp function from Moveable with the inputs IID\_JEWEL and 100 for points. This call will set the object to dead if a player is on the same square and decrease the Jewel count in studentWorld. It also tells the studentWorld (through pickUp) to add 100 points to the score. This function is defined here because Jewels are a unique object derived from Goodie, and picking it up has effects on GoonieBlast that are different from other goodies. It is not pure virtual because it has no derived classes.

12. ExtraLife Class (public Goodie)

* ExtraLife
  + Constructor for the ExtraLife class. It takes in a pointer to the student world, an x and y coordinate to mark its starting position, and a sublevel. It starts with no direction and an ID of IID\_EXTRA\_LIFE. This is defined here because as a derived class of Goodie, it requires certain information in order to be constructed. It is not pure virtual because ExtraLife objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function calls the pickUp function from Moveable with the inputs IID\_EXTRA\_LIFE and 500 for points. This call will set the object to dead if a player is on the same square and tells the studentWorld (through pickUp) to add 500 points to the score and one life to the number of lives. This function is defined here because ExtraLife is a unique object derived from Goodie, and picking it up has effects on GoonieBlast that are different from other goodies. It is not pure virtual because it has no derived classes.

13. Ammo Class (public Goodie)

* Ammo
  + Constructor for the Ammo class. It takes in a pointer to the student world, an x and y coordinate to mark its starting position, and a sublevel. It starts with no direction and an ID of IID\_AMMO. This is defined here because as a derived class of Goodie, it requires certain information in order to be constructed. It is not pure virtual because Ammo objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function calls the pickUp function from Moveable with the inputs IID\_AMMO and 200 for points. This call will set the object to dead if a player is on the same square and tells the studentWorld (through pickUp) to add 200 points to the score and add 25 to the ammo count. This function is defined here because Ammo is a unique object derived from Goodie, and picking it up has effects on GoonieBlast that are different from other goodies. It is not pure virtual because it has no derived classes.

14. RestoreHealth Class (public Goodie)

* RestoreHealth
  + Constructor for the RestoreHealth class. It takes in a pointer to the student world, an x and y coordinate to mark its starting position, and a sublevel. It starts with no direction and an ID of IID\_RESTORE\_HEALTH. This is defined here because as a derived class of Goodie, it requires certain information in order to be constructed. It is not pure virtual because Ammo objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function calls the pickUp function from Moveable with the inputs IID\_RESTORE\_HEALTH and 1000 for points. This call will set the object to dead if a player is on the same square and tells the studentWorld (through pickUp) to add 1000 points to the score and restore m\_hitpts to 20. This function is defined here because RestoreHealth is a unique object derived from Goodie, and picking it up has effects on GoonieBlast that are different from other goodies. It is not pure virtual because it has no derived classes.

15. Bot Class (public Moveable)

* Bot
  + Constructor for the Bot class. It takes in a pointer to the student world, an ID used to identify the object, an x and y coordinate to mark its starting position, a starting direction, a sublevel, and a starting hitpt value. Also, based on the level, it initializes the private member m\_ticks to a specific that will determine how fast the objects will move on the screen. This is defined here because as a derived class of Moveable, it requires certain information in order to be constructed, and must get this information from any derived class that is being instantiated. It is not pure virtual because Bot objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function is a pure virtual function. A definition is not provided here because each object derived from Bot needs to “do Something,” but have different behaviors. Also, it is a pure virtual function because the Bot class should be an abstract class.
* receiveDamage
  + This function takes in an integer parameter, then decreases its private member m\_hitpts by that value. This function is defined here because only Bots receive damage in this way; each of its derived classes share this functionality, which is not unique. It is not a pure virtual function because objects derived from Bot use it in the same way.
* checkKilled
  + This function accesses the private member m\_hitpts and checks whether its value is equal to or less than 0. If these conditions are true, it sets the bot to dead and plays the appropriate sound. This function is defined here because all bots die once their hitpts become less than or equal to zero, and the objects derived from Bot must use this functionality in their doSomething. Also, access to m\_hitpts is made more simple because m\_hitpts is a member of the Bot class. It is not pure virtual because the derived members of Bot use this function in the same manner.
* Rest
  + This function determines whether the appropriate amount of ticks have passed before the bot moves again. It first accesses the private members m\_ticks and m\_rest. If m\_rest is equal to 0, then it resets m\_rest to m\_ticks and returns false. Otherwise, it decrements m\_rest and returns true. This function is defined here because all bots need to rest a certain amount of ticks, and this function allows bots to determine when it should move or rest. Access to private members m\_ticks and m\_rest is more simple than accessing from a derived class. It is not pure virtual because the derived members of Bot use this function in the same manner.

16. KleptoBot Class (public Bot)

* KleptoBot
  + Constructor for the KleptoBot class. It takes in a pointer to the student world, an x and y coordinate to mark its starting position, and a sublevel. It starts facing right with an ID of IID\_KLEPTOBOT, and 9 hitpts. It also initializes m\_distanceBeforeTurning to 3 and m\_storedGoodie to nullptr. This is defined here because as a derived class of Bot, it requires certain information in order to be constructed. It is not pure virtual because KleptoBot objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function first checks if the kleptoBot is dead or if its m\_hitpts are less than or equal to 0. If this is the case, and its m\_storedGoodie is not nullptr, it sets the object in m\_storedGoodie to visible and tells studentWorld to award 20 points to the user, then returns. It then checks the spot it is standing on to see whether there is a goodie; if so, and m\_storedGoodie is nullptr, it will “bookmark” the goodie with a pointer. It then uses the rest function to see whether it is time for the kleptoBot to move; if so, then it checks whether a) it is not already holding a goodie and b) it is standing on a goodie, and then draws a random number from 0-3; if the value is 0, then it “picks up” the goodie by setting it to invisible, then returns. If the value is not 0, it will attempt to move; if it has not yet moved 3 steps in the same direction, it will continue in the same direction, else if it has moved 3 steps or is blocked, it will choose a random direction to move. This function is defined here because only kleptoBots move/behave in this way; no other object/class derived from its base classes pick up goodies or have a set distance before it can move. This is not a pure virtual function because there are no derived classes to use it.

17. SnarlBot Class (public Bot)

* SnarlBot
  + Constructor for the SnarlBot class. It takes in a pointer to the studentWorld, an x and y coordinate to mark its starting position, a direction, and a sublevel. It starts with an ID of IID\_SNARLBOT and 20 hitpts. This is defined here because as a derived class of Bot, it requires certain information in order to be constructed. It is not pure virtual because SnarlBot objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function first checks whether the SnarlBot is dead or has less than or equal to 0 hitpts. If this is the case then it is setDead and returns. Otherwise, it calls the rest function to determine whether or not it should move on this tick. If so, then it checks whether a player is in it’s line of sight, then fires at it, else it continues to move in its current direction or changes direction if there is an obstacle in front of it. This function is defined here because only SnarlBots need to shoot at players or determine their position; its functionality is not shared by others of the Bot class. This is not a pure virtual function because there are no derived classes to use it.
* findObstacle
  + This function finds whether or not there is an obstacle between the SnarlBot object and a player. First, it records its own coordinates, then it uses the detectPlayer function to determine whether there is a player in its line of sight. If so, then it records the player’s coordinates, and checks each square between itself and the player for obstacles by getting a vector of actors on the square and checking their contents. If there is a wall, kleptoBot, Snarlbot, or KleptoFactory anywhere in between, it returns true. Otherwise, if all of the spaces in between are clear, it returns false. This function is defined here because only SnarlBots use this function, and do so in the same manner; other bots and moveable objects need not use its functionality. It is not a pure virtual function because no derived classes use it.
* detectPlayer
  + This function first records the x and y coordinates of itself, then depending on its direction, searches every square from its own location to the border of the map for a player. If a player is found, then it returns a pointer to that player; otherwise, it returns a null pointer. This function is defined here because only Snarlbots need to detect players, and other bots do not use this functionality. It is not a pure function because no derived classes need to use it.
* moveLine
  + This function takes in a direction and, depending on if the SnarlBot is horizontal or vertical, prompts the SnarlBot to attempt to move in its current direction. If it cannot keep moving in this direction (because of an obstacle), then it sets the SnarlBot in the opposite direction so it can keep moving. This function is defined here because only SnarlBots move in a straight line like this; other bots do not use this functionality. It is not a pure function because no derived classes need to use it.

18. Bullet Class (public Moveable)

* Bullet
  + Constructor for the Bullet class. It takes in a pointer to the studentWorld, an x and y coordinate to mark its starting position, a direction, and a sublevel. It starts with an ID of IID\_BULLET. This is defined here because as a derived class of Moveable, it requires certain information in order to be constructed. It is not pure virtual because Bullet objects are created in the same manner and because constructors cannot be virtual.
* doSomething
  + This function first checks to see if the bullet is alive; if not it returns. If it is living, it calls the hitStuff function in case it is allocated on top of an obstacle, in which case it sets itself to dead. Otherwise, it attempts to keep going in the current direction, then checks again whether it is sitting on an obstacle/target. This function is defined here because it presents a unique functionality of Bullet that no other Moveable object does. It is not pure virtual because it does not have any derived classes that will use them differently.
* moveLine
  + This function will move the position of the Bullet object one square forward, depending on its direction. This function is defined here because the bullet uses its functionality in a unique way that other classes derived from Moveable do not. It is not pure virtual because it has no derived objects that will use it differently.
* hitStuff
  + This function checks to see whether there are any other objects on its square. It first checks to see if there is a pointer; if so, it will tell the player to receive damage, then set itself to dead and do nothing else. If there is no player, then it checks for other actors; if it encounters a bot, it will tell them to receive damage and set itself to dead, and if it encounters a wall, it will set itself to dead immediately, and returns true after it is dead. If it has not encountered anything at this point, then it returns false to indicate that it is still alive. This function is defined here because the bullet uses its functionality in a unique way

**StudentWorld**

StudentWorld

* Constructor for the class. It takes in a string that helps to initialize GameWorld, from which StudentWorld is inherited. It initializes the private members (m\_hostages, m\_jewels, m\_subLevel, m\_health, m\_ammo, m\_timeLeft, m\_finishedLevel) to their base/beginning values. This function is defined here because StudentWorld needs a parameterized constructor in order to be created from the GameWorld base class, and its member variables need to begin at a specific value. This is not pure virtual because constructors cannot be virtual.

getCurrentSubLevel

* This function returns the sublevel that the user is currently in by examining the private member m\_subLevel. This is defined here because the StudentWorld needs to know where each actor is according to sublevel, and because it is handling a private member. This is not pure virtual because it is not used by any derived classes.

setCurrentSubLevel

* This function sets the sublevel to a different one by taking in an int parameter and changing the value of private member m\_subLevel. This is defined here because it is making changes to a private member, which does not allow access by outside objects. This is not pure virtual because it is not used by any derived classes.

init

* This function handles file loading; for the current level, the function formats the file name as well as the sublevel file names into appropriate strings. For every file name that is produced, the function calls on the framework to load that level or sublevel; if it is loading a main level, it resets the m\_hostage and m\_jewel count to 0, and if loading the game for the first time, it resets the score and ammo count to 0 while replenishing m\_health to 20 and m\_timeLeft to 1500. Two for loops are then utilized to traverse each of the chars in the file’s map and dynamically allocate new objects, which are pushed into a temporary vector and pointer. Once a sublevel is completed, the temporary vector and pointer are pushed into the StudentWorld’s private vectors m\_actor and m\_player, and the function moves on to the next sublevel. After a main level and all its sublevels are completely loaded, the function prompts the framework to start/continue the game. This function is defined here because it is the only method to call each of the levels and sublevels and load them, and it handles the private member vectors. It is not pure virtual because it is not used by any derived classes.

move

* This function decides what to do while the game is continuing. While there is still time left, the function updates the display text and checks whether or not the player is still alive. If so, then the function goes through the m\_actor vector at the current sublevel and tells each actor to doSomething. The player is then given a chance to doSomething. If the player is dead, then the function decreases the player’s lives, restores it to full health, and returns the GWSTATUS\_PLAYER\_DEAD screen. Any players that have died during this tick are then removed by checking the status of each actor at the current sublevel and deleting them. Otherwise if the level is finished, the function notifies the framework, which updates the level. This function is defined here because it handles private member vectors from the class in a manner that outside objects should not/cannot. It is not pure virtual because it is not used by any derived class.

cleanUp

* This function uses a for loop to first traverse through the private member vector m\_player and delete each of its pointers. After clearing the vector, it moves onto the vector m\_actors; for every subvector inside m\_actors, another for loop is utilized to delete each pointer, and the subvector is cleared. This function is defined here because it handles the private member vectors from the class in a manner that outside objects should not/cannot. It is not pure virtual because it is not used by any derived class.

setDisplayText

* This function outputs a string of text containing information about the current score, current level, current sublevel, life count, health, ammo, and time left. It utilizes stringstream to format the information with the correct amount of spaces, 0’s, -‘s, and labels into a single string that is then displayed through the GameWorld function setGameStatText. This function is defined here because the string it produces uses private member functions from the class. It is not pure virtual because it is not used by any derived classes.

decCount

* This function takes in an integer representing the ID of an object, then subtracts one from its corresponding member variable. This function is defined here because it changes private members whose information needs to be modified/accessed by outside objects. It is not pure virtual because it is not used by any derived classes.

revealExit

* This function returns a Boolean value based on whether or not the private members m\_jewels and m\_hostages are equal to zero; it returns true if both are equal to 0 and false otherwise. This function is defined here because it accesses private members that otherwise cannot be seen by outside objects. It is not pure virtual because it is not used by any derived classes.

addAmmo

* This function takes in an integer and increases the private member m\_ammo by that integer. This function is defined here because it changes a private member whose information needs to be modified/accessed by outside objects. It is not pure virtual because it is not used by any derived classes.

restoreHealth

* This function sets the private member m\_health to 20, which is its maximum value. This function is defined here because it changes a private member whose value depends on the behavior of outside objects. It is not pure virtual because it is not used by any derived classes.

getHit

* This function takes in an integer and decreases the private member m\_health by that integer. This function is defined here because it changes a private member whose information needs to be modified/accessed by outside objects. It is not pure virtual because it is not used by any derived classes.

getHealth

* This function retrieves the value of the private member m\_health and returns it. This function is defined here because it accesses a private member whose information is needed by outside objects. It is not pure virtual because it is not used by any derived classes.

getAmmo

* This function retrieves the value of the private member m\_ammo and returns it. This function is defined here because it accesses a private member whose information is needed by outside objects. It is not pure virtual because it is not used by any derived classes.

makeNew

* This function takes in a pointer to a dynamically allocated object (typically a KleptoBot or Bullet) and pushes it into the private vector m\_actors at the current sublevel. It is defined here because it accesses the private vector m\_actors, whose contents need to be modified according to the behavior of outside classes. It is not pure virtual because it is not used by any derived classes.

bonusTime

* This function retrieves the value of the private member m\_timeLeft and returns it. This function is defined here because it accesses a private member whose information is needed by outside objects. It is not pure virtual because it is not used by any derived classes.

finishedLevel

* This function sets the private member m\_finishedLevel to true in order to indicate that the player can move on to the next level. This function is defined here because it changes a private member that needs to be accessed by outside objects. It is not pure virtual because it is not used by any derived classes.

getContents

* This function takes in two ints representing x and y coordinates representing a location, then checks all of the actors in the m\_actors vector at that sublevel to determine whether any of the actor’s positions match the specified coordinates. If there is a match, then a pointer to that actor is pushed into a temporary vector, which stores all objects at that particular spot. This vector is then returned. This function is defined here because it searches through the private vector m\_actors whose information is needed but cannot be accessed by outside objects. It is not pure virtual because it is not used by any derived classes.

getPlayer

* This function takes in two ints representing x and y coordinates representing a location, then checks the position of the player in the current sublevel to determine whether or not the player is at the specified position. If the player is present at that square, it returns a pointer to it; if not, it returns a null pointer. This is defined here because it deals with the private vector m\_player whose information is needed by outside objects. This is not pure virtual because it is not used by any derived classes.

~StudentWorld

* Destructor for the class, which calls the cleanUp function in order to delete all newly allocated objects. This is defined here because it deletes the members in the private vectors m\_actors and m\_player. This is not pure virtual because it is not used by any derived classes.

**Incomplete functionality and bugs**

* When shooting bullets at a kleptoBot Factory, bullets will pass through them (but will not hit them). This was implemented in order to allow the player to hit kleptoBots standing directly on factories.
* The program will crash on occasion if shooting a large number of bullets at one time (presumably a problem with deletion or cleaning up dead objects), resulting in a malloc\_error bug. This happens very rarely (~1/20 times), so testing and debugging it was very difficult.
* When the player dies, the health in the display text will briefly flash -5% before the GWSTATUS\_PLAYER\_DIED screen appears.
* Due to the difficulty of the game, I was unable to test past main level 1, so functionality after that is unknown.

**Design Decisions and Assumptions**

* I did not make any notable assumptions in writing the code for this project; behavior for all objects is based on the specifications or sample game provided.

**Testing**

* StudentWorld
  + To test this class, I made sure that each level had objects in the correct places, and that for every tick, moveable objects performed some action (even if just standing). When going through gates, I made sure that the sublevel changed correctly, and that when finishing levels, the correct screen appeared and the next level showed up. I wrote cerr statements to declare when a sublevel was and was not loaded. I also wrote cerr statements to ensure that the names of each file being loaded were correct. When moving between levels, I checked that the StudentWorld maintained points and lives but reset health, ammo, and time. When time ran out, the player pressed esc, or the player lost a life, I tested for the GW\_STATUS\_PLAYER\_DIED screen appeared, and the game resumed to the main level.
* Actor
  + To test this class, I made sure that I could not instantiate it as an object by itself. I then checked to see that each of its derived classes were able to use its functions (setDead, getWorld, etc). It passed these tests.
* Moveable
  + To test this class, I checked to see that I could not instantiate it as an object by itself. I then ensured that its derived classes were able to use its functions (tryToMove, go, fire, etc.). It passed these tests.
* Stationary
  + To test this class, I checked to see that I could not instantiate it as an object by itself. It passed this test.
* Goodie
  + To test this class, I checked to see that I could not instantiate it as an object by itself. Also, I tested its functions through its derived classes, each of which used the pickUp function; the pickUp function should have changed the correct values corresponding to the goodies.
* Gate
  + To test this class, I made sure that the Gate was sitting on the correct location and that it did not have any other objects on top of it. When placing the player on the same square as a gate, I made sure that the sublevel changed appropriately and that the game changed to a different interface corresponding with that sublevel, after which I checked to make sure that the gate did not reappear in the same spot (cannot be reentered). It passed these tests.
* Wall
  + To test this class, I made sure that the Walls did not move and were placed in the correct positions. It passed this test.
* KleptoFactory
  + To test this class, I made sure that the KleptoFactory was initialized in the correct location. Then, I made sure that it produced KleptoBots at a reasonable pace (not uncontrollably). I used the player to block KleptoBots from moving, and by keeping them in the vicinity of the KleptoFactory I made sure that it did not produce KleptoBots if at least 3 were near it. I also made sure that bullets could not hit it, and that it produced the correct sound when making KleptoBots.
* Exit
  + To test this class, I checked that the Exit was not visible at the start of every level. I then checked that it did not appear until I collected all jewels and hostages; however, once jewels and hostages were all collected I made sure that it was made visible when returning to the main level and that it made the right sound when appearing. I also made sure that it did not continue to make its “appearance” sound effect as long as it was visible. Then, when the player stands on the same square as the exit, I tested to see that the game showed the Finished Level screen. It passed these tests.
* Player
  + To test this class, I made sure that the player was initialized in the correct location, facing right. I made sure that it responded to key inputs correctly, and that it moved only when appropriate; it should not have moved if there were any obstacles in the way. It passed these tests.
* Hostage
  + To test this class, I made sure that when running the game the hostage faces right and appears in the correct position. Then, when a player is on the same square, it must disappear. If it was not collected I checked that the Exit did not appear after exiting the sublevels; it passed this test.
* Jewel
  + To test this class, I made sure that the Jewels appear in the correct positions and not on top of any other objects. When placing the player on top of it, it must disappear and appropriate 200 points to the player’s score. If not all jewels were collected, I checked that the Exit did not appear. Otherwise if all jewels were collected, the Exit should appear. It passed these tests.
* ExtraLife
  + To test this class, I made sure that the Extra Life goodies appear in the correct positions and not on top of any other objects. When placing the player on top of it, it must disappear and appropriate 500 points to the player’s score. It must also add 1 to the amount of lives displayed in the display string. It passed these tests.
* Ammo
  + To test this class, I checked that the Ammo goodies appear in the correct positions and not on top of any other objects. When placing the player on top of it, it must disappear, grant 200 points to the player, and add 25 to the ammo count displayed in the display string. It passed these tests.
* RestoreHealth
  + To test this class, I checked that the Restore Health goodies appear in the correct positions and not on top of any other objects. When placing the player on top of it, it must disappear, give the player 1000 points, and change the health percentage in the display string to 100%. It passed these tests.
* Bot
  + To test this class, I made sure that I could not instantiate it as an object itself, and that each of its derived classes could use its classes (checkKilled, rest, receiveDamage). It passed these tests.
* SnarlBot
  + To test this class, I made sure that each Snarlbot was initialized in the correct location and not on top of anything else. Depending on its type, each SnarlBot should have moved in one plane only: either only left and right or only up and down. When they encountered obstacles, they should turn the other direction, and if they sense a player, they should shoot. I checked to see that they knew when or when not to shoot by placing my player in front of them and behind obstacles; they were not supposed to shoot if the player was behind a wall. To test this, I wrote cerr statements that stated when there was a player in range, and when there was an obstacle between the SnarlBot and the player, indicating that it would not fire at that moment. I also made sure that when they both shot and were hit, they made the appropriate sounds, and that they died after 10 hits; I checked that the correct amount of points were granted. It passed all of these tests.
* KleptoBot
  + To test this class, I made sure that each KleptoBot was initialized in the correct location on top of a KleptoBot Factory. I checked to see that it did not move when cornered, and that if it was able to move, only moved three squares before changing directions. I also watched kleptobots to make sure that they collected only one goodie, and when they died, that their goodies reappeared in their original places. It passed these tests.
* Bullet
  + To test this class, I made sure that anytime a Bullet object was allocated, it appeared one square in front of the actor that fired it. It must also die when encountering any obstacle (KleptoBot, SnarlBot, Wall, Player), and appropriate the correct amount of damage; I checked this by counting the amount of times a bullet hit a bot before it died, and by monitoring the health of the player as it was hit by bullets. I also checked that if a player shot the bullet onto an obstacle, it was visible but killed itself immediately; it must also not appear on top of obstacles if not shot from an adjacent position. It passed all of these tests.