Rakshit Garg

404 420 830

**1. High Level Description - Actor.cpp**

class Base extends GraphObject

Base (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Constructor that takes in an image ID, x, y, direction, studentWorld pointer, and string for the name of the class. It sets visibility and the object’s alive status to true and assigns the name parameter to a name private variable.)

virtual void doSomething() = 0; (Pure virtual function that is in the most base class because every derived class must have a doSomething() implementation, and each object does something different.)

bool isAlive (); (A getter function that returns a specific class’s alive or dead status. Not virtual because it doesn’t need to be implemented any differently for any class.)

void setAliveorDead (bool alive); (A setter function that sets an object’s alive or dead status—-a parameter of true means that the object is alive. Not virtual because the implementation is constant regardless of class.)

StudentWorld\* getWorld(); (A getter function that returns a pointer to the StudentWorld object that the objects reside in. Not virtual because implementation does not change.)

int getHealth (); (A getter function for health. This is in the most base class because both Actors and Objects have health (i.e. the Player and the Boulder.))

void setHealth (int hp); (A setter for health variable. Not virtual because implementation does not change.)

void changeHealth(int howMuch); (A changer method for the health variable. Used whenever decrementing or incrementing the health of an object. )

virtual bool isBlocked(); (A virtual function for determining if an object blocks the player. This is in the most base class because every object can either block or not block the player.)

virtual bool isAttackable(); (A virtual function for determining if an object can be attacked by bullets. This is in the most base class because every object can be on the same place as a bullet, but they all interact with a bullet in different ways - i.e. taking damage versus letting a bullet pass over them).

virtual void damage(); (A virtual function that contains the logic for damaging a class that has been hit with a bullet. This is virtual because each object is damaged in a different way, in ways such as the amount of points to award the player.)

virtual ~Base() = 0; (Virtual destructor that guarantees the class will be treated as an abstract class. We don’t want anyone creating an object of type Base)

string getClass(); (Getter function for the name variable that was initialized at construction. This returns the name of the string and aids in collision detection. Not virtual because it simply returns the name variable which holds different values depending on the kind of object that it is.)

class Actor extends Base (This class doesn’t do much on its own. Its meant to be a short class that helps make the distinction between the Player/Robots and Objects more clear.)

Actor (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Constructor that passes its parameters to the constructor for Base.)

virtual void doSomething () = 0; (Pure virtual function that makes Actor an abstract class. This will be implemented by every class that is derived from Actor.)

class Object extends public Base (This class doesn’t do much on its own. Its meant to be a short class that helps make the distinction between the Player/Robots and Objects more clear)

Object (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Constructor that passes its parameters to the constructor for Base.)

virtual void doSomething () = 0; (Pure virtual function that makes Object an abstract class. This will be implemented by every class that is derived from Object.)

class Player extends public Actor

Player (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Player constructor that passes its parameters to the Actor constructor and sets the Player’s health to 20 hit points).

virtual void doSomething (); (Every time a key is pressed, this will move the player appropriately by called keyPressMovement(int x, int y, Direction dir), as long as the player is alive. This is virtual because it is redefining the doSomething() method found in its base classes.)

virtual bool isAttackable(); (Returns true because a player can be harmed by bullets. This is a virtual because it redefines the isAttackable() method found in base classes.)

void keyPressMovement(int x, int y, Direction dir); (Moves the player given an x, y, and direction and checks to make sure that there are no obstacles in that direction so the player can actually move. This is not virtual because it is not in any other class, derived or extended.)

virtual void damage(); (Virtual method that defines how the player will be damaged. It changes the health by 2, and if the health falls before 0, it will play a sound that the player died, decrement the amount of player lives, and set the player’s alive status to false. Otherwise, it will play a sound indicating the player was hurt and the game will keep going.)

virtual bool isBlocked(); (Returns true, because players block robots from moving onto its square. This is virtual since it redefines the isBlocked method found in base classes.)

class Robot extends public Actor

Robot (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Constructor that will pass its 5 parameters onto the Actor constructor that it is calling).

virtual bool isAttackable(); (Returns true because a robot can be harmed by bullets. This is a virtual because it redefines the isAttackable() method found in base classes.)

virtual bool isBlocked(); (Returns true, because robots block players from moving onto its square. Additionally, boulders cannot be moved onto the same space as a robot. This is virtual since it redefines the isBlocked method found in base classes.)

int getTicks(); (Returns the amount of ticks that have passed.)

void incrementTicksByOne(); (This will increase the amount of ticks that have passed by 1. This function is called once every tick, and this keeps a real time count of ticks passed in the game. This is nonvirtual and is in Robot because all robots operate on the same tick logic.)

bool linearMove(); (Checks for obstacles in the current direction of the robot. If there are obstacles, it won’t move and will return false. If there aren’t obstacles, it will move and once it has relocated the robot, return true. This is not virtual because all robots will move in the same way if they only need to move once in whatever direction they’re already facing. )

bool moveWithSpecifiedDirection(Direction dir); (Takes in a specific direction and checks for the obstacles in that direction. If there are any obstacles, it won’t move and will return false. If there are, it will relocate the robot to one square in that direction that was passed in and return true. This does not need to be virtual because no other class will redefine this function. All robots move in this way with a specified direction.)

bool isCorrectTick(); (Uses the math found in the spec to calculate whether or not the robot should be doing something during this tick. If it should, the function returns true. If not, it will return false. Not virtual since all robots operate on the same tick logic. There is no tick where only certain kinds of robots can move and others cant.)

bool shoot(); (Checks for obstacles in the bullets path. If there aren’t object’s, it will create a new bullet, play a sound that the bullet has been fired, and add it to the student world. It will then return true. If the bullet was unable to be created because of an obstacle, it will return false.)

class SnarlBot extends public Robot

SnarlBot (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Sets the health of the SnarlBot to 10, and passes its parameters to the constructor of the Robot).

virtual void doSomething(); (Checks to see if a SnarlBot is alive. If it is, it’ll check if the game is currently on the correct tick. If it is, then the snarlBot will try to shoot if it can. If it can’t do any of these things, it will move in the direction it is facing, and if it can’t move, for example, if its boxed in, it will flip directions during each correct tick. Virtual because this is redefining the doSomething that was in Actor.)

virtual void damage(); (Decrements health by 2, and if the health hits 0, it will set the alive status to false, play the sound of a robot dying, and increase the score by 100. Otherwise it will play a sound for the robot being damaged. This is virtual because each robot gives a different kind of points.)

class KleptoBot extends public Robot

KleptoBot (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Sets the distance before turning for the Kleptobot to an integer in between 1 and 6. Also passes in its parameters to the Robot constructor.)

Direction randomDirectionGenerator(); (Creates a random direction and returns it. The derived regular kleptobots and angry kleptobots will inherit this method without needing to change the implementation, so this method does not need to be virtual.)

void setDistanceBeforeTurning(int rand); (Sets the distance before turning variable to the integer that is passed in. Does not need to be virtual since it will just be inherited as is by all of its derived class.)

int getDistanceBeforeTurning(); (Getter method for the distance before turning variable. This simply returns the integer that will be used in comparisons later on to see if the kleptobot should be moving or changing direction. This will also be inherited so it doesn’t need to be virtual since it will never be redefined.)

bool pickedUpAGoody(); (If the kleptobot is on the same square as a goodie, this method will pick up the goodie 10% of the time that it is called. It’ll return true if the goodie was picked up and false if it wasnt. This doesn’t need to be virtual since both kinds of kleptobots deal with goody’s in the same way.)

bool getHoldingGoody(); (Returns whether or not a kleptobot is holding a goody. This is the same for both kinds of kleptobots, which is why its in the kleptobot class and nonvirtual.)

string getGoodyType(); (Returns a string that represents the kind of goodie that the bot picked up. This is constant in its implementation and is the same for both regular and angry kleptobots.)

bool moveWithoutObstacles(); (If there are no obstacles in the way, then this function will call the linearMove function defined earlier and move once in the direction the robot is already facing. Afterwards it will return true. Otherwise, if there is an obstacle or it can’t move for whatever reason this function will return false. This is constant for kleptobots, so it makes sense to be in a base kleptobot class rather than just angry or regular kleptobots.)

bool getFoundObstacle(); (Returns the boolean for foundObstacle. This is used in the moveWithoutObstacles function to see whether or not a movement is possible in that direction. This also does not change depending on the type of kleptobot.)

bool moveWithObstacles(); (If the distance moved is more than the distance before turning, or if we have found an obstacle, this function will reset all of the boolean variables to false and then go through each of the four directions until it finds one it can move in. If it has considered all four, and it couldn’t move in any of them, it will just set the direction of the robot to a random direction until the next tick. If we have set the direction or have moved, this function returns true. Otherwise it will return false. This is the same movement logic for both kinds of kleptobots, so its okay to have it just be inherited and be non virtual.)

class RegularKleptoBot extends public KleptoBot

RegularKleptoBot (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Sets the health to 5, and passes in parameters that will initialize the KleptoBot constructor.)

virtual void doSomething(); (If the kleptobot is alive, and its the correct tick, if the kleptobot picked up a goody this will return. Otherwise, if it can move and there are no obstacles, it will do so by calling the moveWithoutObstacles() method. Otherwise, if there is an obstacle, it will call the moveWithObstacles() method and perform an action that way. This is a specific implementation that only pertains to regular kleptobots, which is why it is a virtual method.)

virtual void damage(); (Decrements the robot’s health by 2, and if the health hits 0, it’ll set alive to false. If the kleptobot was holding a goodie, it will create a new goody where it died. Then this method will play a sound for the robot dying and increase score by 10. If the kleptobot doesn’t die, it’ll just play an impact sound. This is a unique implementation specifically for regular kleptobots, which is why it needs to be virtual and can’t just be inherited from some base class.)

class AngryKleptoBot extends public KleptoBot

AngryKleptoBot (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Sets the health to 8, and passes in parameters that will initialize the KleptoBot constructor.)

virtual void damage(); (Decrements the robot’s health by 2, and if the health hits 0, it’ll set alive to false. If the kleptobot was holding a goodie, it will create a new goody where it died. Then this method will play a sound for the robot dying and increase score by 20. If the kleptobot doesn’t die, it’ll just play an impact sound. This is a unique implementation specifically for regular kleptobots, which is why it needs to be virtual and can’t just be inherited from some base class.)

virtual void doSomething(); (If the angry kleptobot is alive and the game is in the correct tick, the robot will try to shoot if the player is in its line of fire. If it can’t shoot, it will move, without obstacles at first, and if it can’t do that then with obstacles. All of these actions are facilitated by helper methods, so that the doSomething is calling a shoot(), pickedUpAGoody(), moveWithoutObstacles(), and moveWithObstacles(). This is a design decision made for both readability and reusability, and allows the angry kleptobot to preserve most of the logic of the regular kleptobot with the addition of a shooting mechanism.)

class Factory extends public Object

Factory (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name, string type); (Sets the type of factory that it is to the string variable that represents that type. Also, passes in the numerous parameters to the object constructor.)

virtual bool isAttackable(); (Returns false, because a bullet doesn’t do anything to Factories, it just hits it and nothing happens. This is virtual because it redefines the isAttackable function that is found in other classes)

virtual void doSomething(); (If the number of kleptobots in a 3 square radius from the factory is less than three and there isnt a kleptobot of either type on the factory, there is a 1 in 50 chance that a new kleptobot is going to be created. If it is, this will play a sound indicating a new robot has been born. This is a virtual function because it redefines the doSomething method that is found in its parent classes.)

virtual bool isBlocked(); (Returns true, because the player isn’t allowed to move onto factories. This is redefining the isBlocked that is found in other Object derived classes, which is why it needs to be a virtual function.)

int getKleptos(int x, int y); (Retrieves a count of the amount of kleptobots in a 3 square radius. If that radius takes us out of the bounds of the map, this will automatically check for bounds and adjust the radius accordingly. Returns that integer count to be used in createKlepto(). This does not need to be virtual because this is the only place where it is used.)

bool createKlepto(); (If the type of factory is regular, this will create a regular kleptobot and call a student world function to add it to the list of objects in the game. If its an angry factory, it will do the same thing, except with new angry kleptobots. If it is unable to create robots, it will return false. Otherwise it returns true. Because this is a method that only affects the factory class, the function doesnt need to be virtual.)

class Bullet extends public Object

Bullet (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Depending on the direction that the bullet is created with, the bullet will be rendered on the square over one in that direction. Additionally, this constructor passes its parameters into the object parameter.)

virtual void doSomething(); (If the bullet is alive, and the object at its x and y is attackable, then the bullet will damage that object. It will then set itself to false and return. Otherwise, it will keep moving and if the next object is attackable it will damage it and set itself to false. This is virtual because doSomething is being redefined here.)

virtual bool isBlocked(); (Sets isBlocked to true, because the player can’t walk through bullets. It needs to be blocked by them so that it can be hurt by them and not move while its being shot. This is not virtual because it doesn’t redefine the isBlocked function. )

void move(); (depending on the direction of the bullet, it will move over one square in that direction. This is specific only to bullet, which is why its not a virtual function.)

class Exit extends public Object

Exit (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Sets itself to invisible by default and passes its parameters into the Object constructor.)

virtual void doSomething(); (Detects if the player has stepped on it and it is visible. If this is true, it will finish the level, add 2000 to the player’s score, and play a sound to tell the player that they’ve completed that level.)

class Wall extends public Object

Wall (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Passes its parameters into the Object constructor)

virtual void doSomething(); (Walls dont do anything. This method doesn’t do anything. Its virtual because doSomething is in every class and so to redefine it for Walls it needs to be denoted by virtual.)

virtual bool isBlocked(); (Returns true because the player and boulders should not be able to go through walls. This is virtual because it has been redefined from its parent class.)

virtual bool isAttackable(); (Returns true because Walls can be attacked in that bullets hit them and die. They don’t have a health value so this doesnt lead to damage or some external effect.)

};

class Boulder extends public Object

Boulder (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Passes its parameters into the Object constructor, and sets health to 10 hit points.)

virtual void doSomething(); (Boulders don’t do anything. So this just returns.)

virtual bool isBlocked(); (Returns true, because boulders block players.)

virtual bool isAttackable(); (Returns true, because boulders can be attacked and destroyed.)

void push(int x, int y); (Moves itself to its current x plus the x parameter that is passed in, and the current y plus the y parameter that is passed in)

virtual void damage(); (Decrements health by 2, and if it goes below 0, sets its alive status to false.)

class Hole extends public Object

Hole (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Passes its parameters into the Object constructor, and does nothing else)

virtual void doSomething(); (If a boulder is on the hole, this will set the hole’s alive status to false and kill the boulder as well. This is a virtual function because doSomething() is redefined across parent and child classes in the game design.)

virtual bool isBlocked(); (Returns true, because a player shouldn’t be able to step onto a hole, unless its pushing a boulder onto it, that is. This is virtual because it can return different things depending on the type of object across its derivation structure.)

class Goody extends public Object

Goody (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Passes its parameters into the Object constructor, and does nothing else.

class Jewel extends public Goody

Jewel (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Passes its parameters into the Goody constructor, specifically passing in the image id for a jewel)

virtual void doSomething(); (If the player is on the same spot as the jewel, increase score by 50 and play the goodie sound. The goodie is then removed from the map. Virtual because doSomething is redefined. )

class RestoreHealthGoody extends public Goody

RestoreHealthGoody (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Passes its parameters into the Goody constructor, specifically passing in the image id for a restore health goody).

virtual void doSomething(); (If the player is on the same spot as a RestoreHealthGoody, increase score by 500, play a sound that the goodie was taken, set health to 20, and kill the goodie. This is virtual since doSomething here is different from doSomething from its parent.)

class ExtraLifeGoody extends public Goody

ExtraLifeGoody (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Passes its parameters into the goody constructor, specifically passing in the image id for a extra life goody.)

virtual void doSomething(); (If the player is on the same spot as an Extra Life Goody, increase score by 1000, play a sound that the goodie was taken, increase the number of lives, and kill the goodie. This is virtual since doSomething has been redefined.)

class AmmoGoody extends public Goody

AmmoGoody (int IID, int x, int y, Direction dir, StudentWorld \*studWorld, string name); (Passes its parameters into the goody constructor, specifically passing in the image id for an ammo goody).

virtual void doSomething(); (If the player is on the same spot as an Ammo Goody, increase the score by 100 and increase ammo by 20. A sound will play and the goody will be removed. This is virtual since doSomething is being redefined).

**High Level Description - StudentWorld.cpp**

~StudentWorld(); (Iterates through the list of actors and deletes each item)

virtual int init(); (Loads the level. Checks to make sure the level has not yet hit 100, and that the file is not in a bad format and can actually be loaded. If the level file is able to be loaded, it will load the level and create all of the objects that the level needs. This is virtual because it is redefining the init method found in its superclass, GameWorld.)

virtual int move(); (Updates the line of text that displays at the top of the screen during gameplay. It then iterates through the list of active actors, and if each one is alive, it will ask them to do something. If the player is found to be dead, the game will end. If the player has completed the level, the score is increased, the bonus is reset to 1000, and the level is finished. Otherwise, if the actor is not alive, it will be removed from the list. Bonus is decremented by 1 point each time this function is called. Once all of the jewels in the level are picked up, this function will set the exit to be visible and play a sound to show that the exit has been revealed. If the player has died, we return and the game ends. If the level has been finished, the score is increased by the remaining bonus amount, and the bonus is reset to 1000. If none of these things occur, the game simply continues. This is virtual because it is redefining the method found in GameWorld.)

virtual void cleanUp(); (Iterates through the list and deletes each object in it. This is virtual because it redefines the method found in its superclass, GameWorld.)

void updateDisplayText(); (Creates a string to represent the data that needs to be displayed. )

Base\* getObject(int x, int y); (Iterates through the list, and given an x and a y location, returns a pointer to the object at this location. If there is no object, it will return an null pointer.)

Base\* getObjectNotAt(int x, int y, string ID); (Iterates through the list and finds the object at a location at x and y, as long as it does not match the string ID that is passed in.)

StudentWorld\* getStudentWorld(); (Returns a pointer to the student world that is in use.)

int getAmmo (); (Returns the amount of ammunition left that the player has.)

void incAmmo (int howMuch); (Increases or decreases the ammunition by an amount that is passed in as a parameter.)

void addBullet (Bullet\* bullet); (Adds a bullet to the actor list.)

void addKlepto (KleptoBot\* k); (Adds a kleptobot to the actor list.)

void setExitVisible(Exit\* e, bool yesOrNo); (Toggles the exit’s visibility depending on whats passed in as a parameter.)

bool areThereJewelsLeft(); (Iterates through the list and if it finds a jewel, returns true. If there were no jewels found in the list, return false.)

unsigned int getBonus(); (Gets the bonus amount that is left.)

void decBonus(); (Decrements the bonus by 1 as long as the bonus is a positive number, so that it never dips below 0.)

void setBonus(int number); (Sets the bonus to whatever number is passed in as a parameter).

bool getPlayerDead(); (Gets the alive or dead status of the player. True represents that the player is dead, and false represents that the player is alive.)

void setCompletedLevel(bool isHeDone); (Sets the level completed to be true or false, depending on the parameter that is passed in.)

bool getCompletedLevel(); (Gets whether or not the level has been completed.)

bool areObstaclesInBetweenRobotAndPlayer(GraphObject::Direction dir, int x, int y, int px, int py); (Checks to see whether or not there are any obstacles in between the Robot (whose location is represented by x and y) and the Player (whose location is represented by px and py). Depending on the direction, the area we need to search changes, so this method takes the direction into account to determine an accurate search area and then it iterates through those locations in the actor list to determine if there are any obstacles in between the robot and the player.)

bool isObjectInRowOrColumn(GraphObject::Direction dir, int x, int y, string ID); (If the direction is left or right, this will check for any objects of type ID in that row. If the the direction is up or down, this checks for any objects of type ID in that column. Returns true if there is an object and false if there isn’t.)

bool isBotFacingPlayer(GraphObject::Direction dir, int x, int y, int px, int py); (Depending on the direction, this finds whether or not the player and the robot are in the same row or column and are facing each other. If they are, this will return true, and if otherwise it will return false.)

int getPlayerX(); (Gets the x value of the player’s location.)

int getPlayerY(); (Gets the y value of the player’s location.)

void createGoody(int x, int y, string type); (Creates a goody of the type that is passed in at the x and y value.)

string levelLoader(); (Creates a string that changes depending on the level number, so that the correct level is loaded each time the player finishes or restarts a level.)

**2. Functionality**

To my knowledge, my code is complete. I do not think that I haven’t implemented any part of the spec.

**3. Design Assumption**

I assumed that boulders cannot be pushed onto the exit square and that Kleptobots should be able to walk over the exit even if it is visible. I also assumed that Kleptobots cannot pick up jewels, they can only pick up goodies.

**4. Testing**

**Actor.cpp**

class Base : public GraphObject

Because this is an abstract base class, I did not do any testing. Instead, I tested its derived classes.

class Actor : public Base

Because this is an abstract base class, I did not do any testing. Instead, I tested its derived classes.

class Object : public Base

Because this is an abstract base class, I did not do any testing. Instead, I tested its derived classes.

class Player : public Actor

To test player, I first made sure that it moved the way it was supposed to, so I placed it in an empty map with only walls and made sure it couldn’t move past them. Afterwards, I incorporated elements such as holes and factories and made sure I couldn’t move past them, but that I could if I was pushing a boulder into a hole. To test its damage function, I stood in the line of fire of a snarlbot and made sure it took 10 shots to hurt me.

class Robot : public Actor

Although Robot is a base class, it contains a few key methods that all robots employ, so this class required multiple rounds of testing. First, I created a SnarlBot that inherits the getTicks and incrementTicksByOne methods and made sure that the SnarlBots were only moving once every so often, and that they were all moving at the same time as each other. Then, I incorporated linearMove and moeWithSpecifiedDirection which allows the snarlbots to move while accounting for no direction, obstacles, and a specified direction. I made sure that the robots would switch direction when they moved or found an obstacle, and recorded to the console in each method with just 1 snarlbot on the screen so i could make sure the methods were being called when they were supposed to. To test shoot, I put the snarlbot behind obstacles such as walls and boulders to make sure it wouldn’t shoot if I (the player) was behind an obstacle.

class SnarlBot : public Robot

Snarlbot did not really need any testing outside of the testing I mentioned in Robot because the methods its using in its doSomething are literally inherited from Robot without any redefinition. In its doSomething, I tested its linearmove by making sure that it would flip direction if the method returned false. This enables the Snarlbot to face the opposite way when it hits a wall such as in Level 00. I needed to test damage by making sure that the appropriate sounds played, so I would shoot the snarlbot 4 times, making sure that each time a hit sound was played, and then on the 5 time, make sure that the dead sound played and the bot was removed from the game.

class KleptoBot : public Robot

DISCLAIMER: Because of the nature of the Kleptobot is so reliant on randomness, there is a degree of uncertainty to my tests.

I spawned a kleptobot and surrounded it with goodies of various kinds so I could make sure that it would only pick up 1 goody at a time and that the goodies would disappear if they were picked up. Also, I made sure there were no other actors in the game and just watched the kleptobot run randomly. Because every other square in the map was a goody, with the exception of a walled border, I could virtually guarantee that a goody should’ve been picked up, since theres a 10% chance of it happening. Then, I spawned a player and killed the kleptobot, so I could verify that the kleptobot had dropped the goody and that the new goody was pickupable by the player. For the kleptobot’s movement functions, I spawned a kleptobot and surrounded it with walls. This let me make sure that it wouldn’t cross over obstacles and it would always set a random direction.

class RegularKleptoBot : public KleptoBot

A regular kleptobot is essentially the same as its base class, except that it implements the functions in Kleptobot. Thus, testing Kleptobot was sufficient to test regular kleptobot.

class AngryKleptoBot: public KleptoBot

An angry kleptobot is also already accounted for since it is the same as a regular kleptobot, except that it can shoot. Thus, testing the shoot function of the robot class and testing the kleptobot class was good enough to test an angry kleptobot.

class Factory : public Object

To test factory, I placed a factory where it needs to go in Level 00, since it is caged by a wall on 3 sides. This lets us make sure that the robots are only created in empty space or on the factory square. I outputted the number of kleptobots in a 3 square radius of the factory to the console every time the doSomething method was called which let me make sure that the code was doing a reliably accurate job of keeping track of how many kleptobots there were around it. With this information, I was able to make sure that the robots where only being born if there were less than 3 around.

class Bullet : public Object

To test bullet, I shot at walls, boulders, and robots of both kinds both with obstacles in between and holes to make sure that the bullets were passing over holes and goodies and jewels but stopping at object such as walls and boulders. Additionally, I made sure that the bullets were damaging the objects appropriately by shooting each object that can be damaged slowly and outputting their health in between. For example, I stood and shot at a boulder slowly until it died, verifying that its health went down by 2 in between each shot, and that it died when the health went below 0.

class Exit : public Object

I ran the level with various amounts of jewels and ran through them to make sure that the jewel counter worked properly, because the correct operation of that counter is essential to exit showing up at the right time. I outputted the number of jewels after each tick of the game, and when it went to 0, verified that the reveal exit sound played and the exit was rendered where its supposed to be. Then, I made sure that the exit was only working for the player, and it wouldn’t finish the level when a kleptobot stood on it, by placing a factory near the exit and waiting for a kleptobot to walk on it.

class Wall : public Object

Walls don’t really do anything so I didnt need to test this very much. I just made sure that the walls blocked objects from going through them by putting the player and making it hit a wall in all four directions.

class Boulder : public Object

Boulders can be pushed, but not onto walls or robots, so I pushed a boulder in all four directions into various obstacles and made sure that it would stop and wouldn’t keep going.

class Hole : public Object

Holes don’t do much. I just made sure that when a boulder was pushed onto the hole, it disappeared, and that bullets could go over holes by shooting at them. I also made sure that robots couldn’t walk over holes by creating a horizontal snarlbot that was surrounded by a hole in both directions.

class Goody : public Object

Goody is a base class that doesn’t do anything on its own. No testing required.

class Jewel : public Goody

I made sure that the points were only rewarded once a player stepped onto the jewel and that other robots couldn’t make the sound play by stepping onto it. I also made sure that the jewel disappeared once it was taken by the player.

class RestoreHealthGoody : public Goody

Testing for this was very similar to the jewel, with the exception of kleptobots being able to take this goody and the other goodies below. This is explained more in detail in the Kleptobot section. I mainly had to add another check that made sure points weren’t rewarded to the player when a kleptobot stole the goody. In terms of the player, I had to make sure that the player’s health went back to 100% when the player stepped on it.

class ExtraLifeGoody : public Goody

I made sure that when the player stepped onto the goody, a life would be added, and that the goody disappeared.

class AmmoGoody : public Goody

I made sure that when the player stepped onto the ammo goody, more ammo would be added and the goody would disappear.

**StudentWorld.cpp**

Because so many of student world’s functions are used by the functions in Actor.cpp (such as isBotFacingPlayer(), getObject(), getObjectNotAt()) testing those classes incorporated tests for many of these functions.

The best way I tested the game was playing it several times and making sure that everything worked normally and there weren’t any outstanding bugs.

In terms of underlying bugs or issues with my code, I specifically tested the leveling mechanism.

Leveling was very important in studentworld, so I made sure that my string stream worked and generated the correct level strings that would be loaded. I made bogus levels and tried to run levels that didnt exist to make sure that appropriate error messages would be shown. I also beat the first level and made sure the second level would show up afterwards.

To test isObjectInRowOrColumm(), I ouputted the return value and placed the player in various cases that I knew would be true or false and matched the output to what I thought the result should be.

To test areObstaclesInBetweenRobotAndPlayer(), I pushed a boulder in between me and a snarlbot, and made sure that the output value of the function was true.