1) Descriptions of each of my public member functions in each of my classes:

**StudentWorld public member functions**

bool isWithinRadius – this method takes in 2 sets of (x,y) coordinates and another distance parameter. If the two sets of coordinates are within the given distance parameter (Euclidian distance), then the function will return true; in other words, if the two coordinates are within the distance parameter, the function will return true, and it will return false otherwise. This is useful because other functions within StudentWorld and Actor objects can call this method to determine whether or not a set of coordinates is too close to another Actor, etc.

bool isCollidingWith – this method is a more specific compared to isWithinRadius. All it does is call isWithinRadius, but it passes in an x,y, and an Actor\*. It then passes in to isWithinRadius the x,y parameters, the actor’s x,y coordinates, and a distance of 3.0. This function returns true if an Actor is within a distance of 3.0 to a set of coordinates, and false otherwise. This is useful because it checks whether or not objects collide with other objects in the game, IE when a boulder hits the Frackman. Furthermore, goodies can only be grabbed when the Frackman is within a distance of 3 from the object. Method returns true if the Actor is within a distance of 3 away from a given set of x,y coordinates, false otherwise.

bool canFrackmanMoveHere – this method checks to see if there are boulders in the 4x4 box that starts with the bottom left corner of the box given by the parameters x and y. This method is called by the Frackman class. If there are no boulders in the 4x4 box given by the x and y parameter, then the function returns true, because the Frackman will be able to move to that coordinate. The function returns false otherwise.

bool canProtestersStepHere – this method is a little different than canFrackmanMoveHere because while Boulders are the only Actors that block the Frackman’s path, the Protesters are blocked by both Boulders and Dirt. This method checks whether or not there exists boulders or dirt within the 4x4 box given by the x and y coordinates. This method also takes in a Direction parameter. If the passed in parameter is right, then the method will check if the x+1,y box is clear of Dirt or Boulders. Method returns true if the box is free of dirt and free of boulders, and false otherwise.

bool isThereDirtHere – this method checks an exact x,y coordinate (1x1 box) and checks if there exists dirt in that coordinate. Returns true if there is dirt, and false otherwise. This is useful to tell whether or not the Frackman should be digging, and whether or not the Frackman should clear the dirt and make the digging noise.

bool isThereBoulderInThisBox – this method checks the 4x4 box given by the x,y coordinate and checks if there exists any boulders in the box. This is useful for telling whether or not a Protester can step in this coordinate. Also, if there is a boulder in this box, then the Frackman will not be able to step into this box. These are only a few examples of when this function is useful. Method returns true if any boulder or part of a boulder exists in the given box and false otherwise.

bool isThereDirtInThisBox – this method checks the 4x4 box given by the x,y coordinate and checks if there exists any dirt in the box. This is useful for telling whether or not a Protester can step in this coordinate. Also, if there is dirt in this box, then a water pool will not be able to form. These are only a few examples of when this function is useful. Method returns true if dirt exists in the given box and false otherwise.

bool clearOfDirtBoulder – this method is a combination of the two preceding methods. It calls the two methods isThereDirtInThisBox and isThereBoulderInThisBox, and only returns true if both methods return false. This is useful for the other Actors to use because it will allow it to check whether or not it can move to a certain location. For example, Squirts and Protesters cannot traverse a coordinate that is already occupied by dirt or boulders.

bool isThereBoulderUnderMe – this method specifically checks if there are any other boulders in the 4x4 box exactly under the box represented by the x,y coordinates passed in. This is only called when a boulder is in its falling state; if the boulder senses another boulder under it, then it will return true. Otherwise, the function will return false.

bool isWithinShoutingDistance – this simply calls isWithinRadius but passes in Frackman’s coordinates and a distance of 4. Since the Protester class does not have access to Frackman’s coordinates, this method is put in the StudentWorld class so that frackman’s coordinates can be passed through. This method is useful because Protesters will know when it is able to shout at the Frackman and when it is not able to shout at it.

bool isFacingFrackMan – this checks whether or not the x,y coordinates and the direction parameter result in a Protester facing the Frackman. For example, if the x coordinate of the Protester is less than the x coordinate of the Frackman (the Protester is to the left of the Frackman), and if the Protester is currently facing right, then you know that the Protester is currently facing the Frackman. Method returns true if Protester is facing the Frackman, and false otherwise.

bool isInLineOfSight – this is also a method made for Protesters. If the Frackman’s x coordinate is exactly lined up with the Protesters and there is no dirt or boulder in the 4x4 box that is drawn from the Frackman to the protester, then the method will return true. A for loop is used to loop through each “step” that brings the protester closer and closer to the Frackman. If any boulders or dirt lie within this path, then the method returns false, because then the Protester would not be in line of sight of the Frackman.

int annoyAProtester – This method takes in a x,y coordinate and a char that represents the cause of being annoyed. Protesters can be annoyed by Boulders, Squirts, and Gold. The method first loops through the vector of Actors and if it reaches any protester, it checks to see if the x,y coordinates are colliding with (within distance of 3) that object at the x,y coordinates. If both are true, then it will annoy the protester and the result will be dependent on the char that is passed through. Boulders completely annoy all protesters, gold causes either staring or immediate annoyance, and squirts will stun protesters until they die, causing them to leave the field. These conditions are all specified by the char, and the char will result in different outcomes. This method returns 0 if it does not successfully annoy any Protesters, 1 if it successfully annoys a Regular Protester, and 2 if annoys a Hardcore Protester.

int min – takes in two ints and returns the int of smaller value. Useful for a lot of the game rules, such as how many actors should be on a field on a given level.

int max – takes in two ints and returns the int of larger value. Useful for game rules, such as the maximum number of protesters that can be on the field at once.

int randInt – takes in two numbers, and returns a random int that is between (inclusive) the two parameters. The parameters can be given in any order, and the method will still work.

void dropNugget – this is a method that adds a Nugget into the Actor vector and gives it specific parameters so that the nugget knows what state to be in. This method is called by the Frackman whenever it has enough gold and can drop a nugget. It will initialize a nugget and will put it in its temporary state at the given x,y coordinates passed into the function.

void reduceBarrels – all this does is decrement the private data member numBarrels. It is called by the Frackman whenever it collides with a Barrel Actor.

void decrementProtesterCount – this decrements the private data member that keeps track of the number of Protesters on a field. It is called by the Protester class whenever a Protester dies and leaves the field.

void annoyFrackMan – this method is called by the Boulder class and the Protester class. Frackman can only be annoyed by boulders and Protester shouts, and will react accordingly. The method takes a char parameter that tells the cause of the annoyance.

void destroyDirt – this method destroys and deletes the dirt that is in the x,y coordinate passed in as a parameter. This is called by the Frackman when it digs through the dirt. It is also called in the Boulder constructor because it needs to be initialized on a spot where dirt does not exist.

void revealCloseObjects – this method is called by the Sonar class to reveal all Objects within a Euclidian distance of 12 units. It calls the isWithinRadius class and passes in 12.0 in its last argument.

Direction faceTheFrack – this method relates the x,y coordinates passed in to Frackman’s coordinates, and returns a Direction that points to the Frackman. For example, if the x value passed in is less than the Frackman’s x value, then it needs to return right to face the Frackman.

Direction getRandDir – this method calls the randInt method and returns a random Direction.

Direction canTurn – this method takes in a x,y coordinate and a direction. If the direction passed in is up or down, it will check if the x,y coordinates to the left and right are open. If both the left and right are open, it will choose which Direction to return randomly. If only one perpendicular direction is available, it will return that direction, and if none of the Directions are viable, then it will return none.

Direction protesterGiveUp – This method takes in two x,y coordinates as parameters but passes them in by value. It will check the grid data member (a 2-D array of Directions that points to the fastest, viable way to the coordinate 60, 60). Depending on what Direction lies in the array at the specific coordinates, it will return the Direction stored in the grid data member. This allows the Protester to navigate the oil field without running into boulders or dirt.

Direction getIntimateWithFrack – this method does the same as protesterGiveUp but instead checks the frackGrid data member which is an array of Directions that point to the fastest route to the Frackman’s current position. This is called by the Hardcore Protester when it is close to the Frackman. Returns the direction that allows it to reach the Frackman in the least amount of steps.

Frackman\* getFrack – returns the Frackman pointer data member. Is only called by Boulder. The Boulder needs to check if it is close to the Frackman so it knows when to kill the Frackman. However, the Boulder class does not have access to the Frackman within its scope, so it needs to go to StudentWorld to get information about the Frackman.

**Actor public member functions**

virtual void doSomething = 0 – This method is a pure virtual method that is defined by its derived classes. Each Actor must doSomething during every tick, but they do different things, so that is why it is a pure virtual function.

isStillAlive – This method returns the private data member in Actor that represents whether or not that specific actor is alive or dead.

virtual void setDead – This method sets the private data member in Actor m\_alive to false, signifying that the Actor has died.

virtual bool doesThisBlock – This method is virtual because when StudentWorld iterates through the vector of Actors, it will ask if that actor -> doesThisBlock. doesThisBlock is a way for the World to differentiate between what Actor it is dealing with. doesThisBlock is defined to return false as default in the Actor class; However, in the Boulder class, doesThisBlock will return true because it is an Actor that blocks movement.

virtual int isProtester – Similarly, this method is virtual so the StudentWorld will know what Actor it is dealing with when it iterates through all the Actors. It returns 0 as a default in the Actor class because most Actors are not Protesters. The isProtester in the Protester class returns 1 and isProtester returns 2 in the HardcoreProtester class.

virtual void getAnnoyed – I want to be able to iterate through each actor and ask if it got annoyed and see its return value. This tells me many things. It tells me what kind of Actor I’m dealing with (returns 0, 1, and 2 depending on what it is) and allows me to annoy an Actor based on what type of Actor it is. It acts as a universal way for the StudentWorld to annoy an Actor by taking in a char parameter, allowing the Actor to react accordingly.

**Dirt public member functions**

virtual void doSomething – This method does nothing. However, it has to be declared because it was declared as a pure virtual function in Dirt’s base class.

**Living Actor public member functions**

virtual void doSomething = 0 – This method is pure virtual because Living Actor is a base class to more specific Actors that have a doSomething. It does not make sense for Living Actor to have a doSomething, but it makes sense for its derived classes to have a doSomething.

StudentWorld\* getWorld – This returns the StudentWorld\* private data member that holds a pointer to the StudentWorld. It gives the LivingActor and all of its derived classes access to public member functions of the StudentWorld class.

void reduceHP – Since all Living Actors have a health component to their gameplay, this public method will be able to be called by all of its derived classes, and will reduce the private data member that represents the object’s HP and will decrement it by a value passed in to the method.

int getHP – This returns the private data member that represents the health of the object, and is available to all of LivingActor’s derived classes.

**Object public member functions**

virtual void doSomething = 0 – This method is pure virtual because Object is a base class to more specific Actors that have a doSomething. It does not make sense for Objects to have a doSomething, but it makes sense for its derived classes to have a doSomething.

StudentWorld\* getWorld – This returns the StudentWorld\* private data member that holds a pointer to the StudentWorld. It gives the Object class and all of its derived classes access to public member functions of the StudentWorld class.

FrackMan\* getFracker – This returns the Frackman\* private data member that holds a pointer to the Frackman. It allows all Objects to call Fracker’s methods, which is useful because Objects directly interact with the Frackman.

**FrackMan public member functions**

virtual void doSomething – This method is defined by the specs of what a Frackman does during each tick. This method should define when the Frackman should turn, shoot, and die based on interactions with the surrounding objects in the oil field. It also destroys dirt when it needs to, and will do nothing if the Frackman is already dead.

int getSquirts – This returns the number of Squirts left in his inventory. Is called once by StudentWorld so that it can update the text display at the top of the game.

int getsCharges – This returns the number of Sonar Charges left in his inventory. Is called once by StudentWorld so that it can update the text display at the top of the game.

int getGold – This returns the amount of GoldNuggets left in his inventory. Is called once by StudentWorld so that it can update the text display at the top of the game.

virtual void getAnnoyed – This was virtual because it was already declared as virtual in Frackman’s base class, Living Actor. Depending on the char that is passed in, it will change Frackman’s alive or dead state, or reduce Frackman’s hit points.

void addSquirts – This method adds 5 squirts to Frackman’s inventory, modifying Frackman’s private data members. It is called by the Water Pool class everytime the Frackman steps on a water pool.

void addCharges – This method adds 1 sonar charge to Frackman’s inventory, by incrementing the private data member m\_sCharges in the FrackMan class.

void addGold - This method adds 1 gold nugget to Frackman’s inventory, by incrementing the private data member m\_gold in the FrackMan class.

**Boulder public member functions**

virtual void doSomething – This method is defined by the specs of what a Boulder does during each tick. The Boulder has three states: a falling, temporary, and sitting state. The method will check to see what state the Boulder is in and act accordingly. If the Boulder is in a falling state, it will continue to check under the Boulder to see if it collides with Protesters, dirt, the lower bound of the oil field, or other Boulders. If any of these hold true, the Boulder will disappear from the oil field. If the Boulder is in a temporary state, it will stay in place a bit before falling. A Boulder can only enter a temporary state if all the Dirt directly below a Boulder has been cleared. If a Boulder is in its waiting state, it will just sit there until it senses that Dirt is being cleared from under the Boulder.

virtual bool doesThisBlock - This method is virtual and was first declared in the Actor class. Since Boulders hinder movement, this method simply returns true and is called in StudentWorld when it iterates through all the actors. This method allows StudentWorld to know when it iterates and is dealing with a Boulder.

**Squirt public member functions**

virtual void doSomething – This method is defined by the specs of what a Squirt does during each tick. This method will check to see if the Squirt is alive. If it is, it will continue to check the space in front of its direction until it hits the wall, dirt, Boulders, or Protesters. If it reaches its max length of 4, it will disappear. If it hits a Protester, it should stun the Protester and set itself to dead. The Squirt will continually check to see if it is able to move or interact with other game objects.

**Barrel public member functions**

virtual void doSomething – This method is defined by the specs of what a Barrel does during each tick. A Barrel sits under the dirt until it is discovered. Once taken, it will notify the StudentWorld that a barrel has been acquired and it will set itself to dead.

**Gold Nugget public member functions**

virtual void doSomething – This method is defined by the specs of what a Gold Nugget does during each tick. It will sit there until it is discovered, then become visible. If it is within three units of Frackman, it will be collected and stored into Frackman’s inventory. It can now be dropped back onto the field, and Protesters can now be bribed by the Gold. However, once dropped, the Nugget has a tick life and will disappear after its life is over.

**Sonar Kit public member functions**

virtual void doSomething – This method is defined by the specs of what a Sonar Kit does during each tick. Sonar Kits just sit in the top left corner if it gets inserted into the map. Once it gets taken, it will disappear from the field. Sonar Kits also have a tick life, and will disappear once the timer reaches 0.

**Water Pool public member functions**

virtual void doSomething – This method is defined by the specs of what a Water Pool does during each tick. Water pools, once constructed, just sit there every tick until either its tick life has reached 0 or if it gets picked up by the Frackman. Once picked up, it will notify the Frackman class that 5 Squirts have been picked up.

**Protester public member functions**

virtual void doSomething – This method is defined by the specs of what a Protester does during each tick. First, the Protester checks to see if it is alive. A Protester has a rest tick counter, and will do nothing during its rest ticks. If the Protester is in a leave the field state, then it will find the best path to the exit (60, 60) and be removed from the field. If the Protester is within 4 units of the Frackman, it will yell at it. However, after yelling, it cannot yell for the next few non-resting ticks. If the Protester is within a direct line of sight of the Frackman, it will change its direction and move one step towards the Frackman. Otherwise, it will decrement its numSquaresToMove private data member by one. Once this counter reaches 0, it will pick a random viable direction and move in this new direction. If the Protester is at an intersection and it hasn’t turned in the last 200 non-resting ticks, then it will turn and pick a new value for numSquaresToMove. Then the Protester will try to take a step in its current direction. If it cannot take a step in its current direction, it will simply set the numSquaresToMove private data member to 0 so that it will pick a new direction during the next non-resting tick.

virtual void getAnnoyed - This method takes in a char parameter that tells the method what is currently annoying the Protester. The method will do something accordingly: When a ‘G’ is passed in, it knows to set the Protester’s leaveField private data member to true so that the Protester will immediately leave the field after being bribed. If the char is ‘S’, the method knows to reduce the Protester’s HP by 2 and to completely annoy it if its HP reaches 0, etc. This method is called by many classes whenever it wants to annoy the Protester.

virtual int isProtester - This returns 1, and is virtual because it was declared as virtual in the Actor class. This allows the StudentWorld and other classes to know when it reaches a Regular Protester when it iterates through the vector of Actors. However, if the Protester is in its leaveField state, it cannot be annoyed, and cannot interact with other objects, so this method actually returns 0 when a Protester gets completely annoyed. This is useful because a Protester cannot be squirted after it starts leaving the field, so StudentWorld won’t annoy the Protester when it is in this state.

void setLeaveField - This method takes in a bool and just sets the private data member leaveField equal to the parameter. This is public because the Hardcore Protester is a derived class of Protester and it needs to call this in order to set the data member, because the Hardcore Protester does not and should not have direct access to the Protester’s private data member.

bool getLeaveField - This method simply returns the private data member leaveField. This is public because the Hardcore Protester is a derived class of Protester and it needs to call this in order to get the data member, because the Hardcore Protester does not and should not have direct access to the Protester’s private data member.

void setTickCounter(int N) - This method sets the private data member tickCounter equal to the int N that is passed through. This is useful because the Hardcore Protester and Protester both need to set its tickCounter equal to a value after it shouts so it freezes.

void setTickCounter() - This method is an overloaded function, similar to the function above. However, it does not take in a parameter because it just sets the tickCounter data member equal to the default value defined in the specs that depends on the level. It is called in the constructor of both the Protester and the Hardcore Protester to set the default tick value before it starts being asked to doSomething.

bool normalMove1 - This method puts all of what BOTH the Protester and Hardcore Protester have in common in their respective doSomethings. No matter what, both the protesters do the same thing up to a some point in the doSomething, so this method is a simple public function that is called by both Protesters in their doSomethings. However, this is bool because if something in this first half gets executed, then everything else in the doSomething for both Protesters must not do something (you cannot do two actions in a single tick of your game). For example, if its leaveField data member is true, then the Protesters must immediately stop what they are doing and walk to the exit. Nothing else should be executed. If two things could happen at once, Protesters might get confused and start walking towards the Frackman when he is in sight.

void normalMove2 - This is the second half of what BOTH the Protester and Hardcore Protester have in common in their doSomethings. This is everything that happens AFTER the one difference that the Hardcore Protester has that the Protester doesn’t have, which is the condition that if the Hardcore Protester is within a certain amount of steps and can reach the Frackman in those steps, then it will chase the Frackman. This is called by both the Protesters. For the Hardcore Protester, normalMove2 will ONLY execute if normalMove1 returns false AND if it is out of chasing range of the Frackman. If these conditions hold, then it will execute the rest of normalMove2. This is a public member function of Protester because Hardcore Protester needs to call it. Both normalMove1 and normalMove2 prevent code duplication, and this code implementation takes advantage of the fact that Protesters and Hardcore Protesters do many similar actions.

**Hardcore Protester public member functions**

virtual void doSomething - This method is defined by the specs. It is exactly like the doSomething of a Regular Protester except for one step in the middle. Therefore, here is how my Hardcore Protester doSomething is structured. If normalMove1 returns false, then get a Direction from the Direction map that is a 2-d array of Directions that point to the best way to get to the Frackman. If that Direction is not none, and the Hardcore Protester is not within shouting distance, then the Hardcore Protester will take a step in this given Direction, and reset its tickCounter, and then return. Otherwise, it will just call normalMove2 of the Protester class and finish up its doSomething. This can be written like this because the Regular Protester and the Hardcore Protester share many characteristics in their respective doSomething methods.

virtual void getAnnoyed - This method is virtual because it was defined by the Actor class as a virtual function. It takes in a char parameter and, based on what char is passed in, will annoy the Hardcore Protester in different ways. Since Hardcore Protesters get annoyed in a similar fashion as Regular Protesters, this is the structure. If the char is G, it will just stare at the gold and the tick counter gets set to a value specified in the specs. Otherwise, if the cause is S (a Squirt) AND the squirt results in the death of the Hardcore Protester, then its HP gets reduced, it will leave the field, play the annoyed sound, set tickcounter equal to 0, and increase the score of the game by 250. Otherwise, if none of the above conditions are met, it will just call the Regular Protester’s getAnnoyed to finish up with condition checking, because they both react to getting annoyed the same way.

virtual int isProtester - This method is similar to the Regular Protester’s isProtester; however, it will return 2 instead of 1. This method is a virtual method because it was defined by the Actor class to be virtual. During run time, when the StudentWorld iterates through the vector of Actors, it will know what kind of Protester it is dealing with when it calls this method, because it knows that it needs to call the isProtester of this class rather than the isProtester of the Actor base class. This is a key concept of Polymorphism and this is a handy trick that allows for a container of many different objects.

2) I was able to implement everything listed in the spec.

3) The spec was a bit ambiguous, so I decided on some functionalities myself. For example, the game in the sample executable shows that after a Protester yells, it stands still for a while, becoming immobilized and unable to yell again AND unable to move again for a couple of ticks. To imitate the sample executable, I decided to immobilize the Protester for 45 non resting ticks after it shouts, so that it stands still. Once it becomes mobilized, it returns to normal and the tick counter gets reset to its default value.

Also, since the Squirt sound does not work in the sample executable, I wasn’t sure if the Squirt sound should go off when an UNSUCCESSFUL Squirt happens. For example, if the Frackman is standing right up against dirt and facing dirt, a Squirt image should not show up because there isn’t enough room for a Squirt to exist in the field. However, in the sample executable, the number of Squirts in Frackman’s inventory goes down, so even though an image doesn’t show up, the Squirt count should go down. Given this information, I am making the assumption that a Squirt sound should go off too, assuming that every time the Squirt count goes down, the Squirt sound should go off too. This also applies to the Frackman when it is right up against the boundary and facing the boundary; a Squirt object should not be created but I designed my game so that the Squirt sound still goes off and the Squirt count still goes down.

4) In order to test all my classes, I often put many cout statements within the code to make sure that certain loops were running and certain variables were incrementing and decrementing when they were supposed to. Specifically, here is what I did to test each class:

**StudentWorld**

I started the game and made sure that all the objects were being created and destroyed when they were supposed to be. To test the functions within StudentWorld such as isWithinRadius, I tried to initiate a world with 30 boulders. Even though their coordinates are random, I wanted to see if it would still space each boulder out so they weren’t too close to each other (greater than a Euclidian distance of 6 from all other game objects other than dirt). To test my cleanUp function, I would press new game or die on purpose to see if everything was successfully deleted and reinitialized for the next level. To test the interactions between Actor objects, I would run around and make sure that everything interacted correctly. I made sure the Frackman would delete dirt and that it would die when a Boulder hits it. I also tried adding 100 Protesters to make sure that they would be able to navigate the maze with the same, shared path finding algorithm. I also made sure every button of Frackman would work and interact. For instance, the StudentWorld must reveal all objects within a distance of 12 of the Frackman whenever the z button is pressed and there are sonar charges left. I tested most of these by initializing Frackman with over 100 items in his inventory so I could repeatedly test the items and their functionalities.

**Actor**

This is not an object specifically in the game, but since all the other classes call the Actor constructor and they are successfully constructed, I know that my Actor class is working the way it is supposed to be working. I tested each virtual function by commenting them out and trying to compile them. Since the derived classes rely on the pure virtual functions and virtual functions in the Actor class, I know that it will result in a compiler error. I also hard coded setDead to see if I could call Actor’s class and set any derived class’ object to be dead. Once an object is set to dead, the StudentWorld will remove the object from the game during the end of the current tick.

**Dirt**

This class was easy to be tested. As long as the game started with all the dirt in the right place, then the construction works. I tested to see if Frackman could dig by running around and making sure that only the dirt in the 4x4 box of Frackman gets deleted. To test the interaction with Protesters and dirt, I would kill a lot of Protesters at once and see if they took the optimal path while avoiding the dirt.

**Living Actor**

I tested this class by constructing its derived classes, the Frackman, Protester, and Hardcore Protester. Since they construct by calling the Living Actor constructor, I know that the Living Actor class constructor is working properly. I tested the reduceHP and getHP by hard coding it and seeing if the Actor would die after its HP reaches 0. By testing its derived classes in more detail, I will be able to decide whether or not the Living Actor class as a whole is working as it is supposed to.

**Object**

I tested this class by constructing the derived classes, such as Squirt and Boulder. As long as these classes construct and work properly, the Object class will work properly, because the derived classes call the constructor of the Object class.

**Frackman**

I tested this class by running around on the map and making sure that it interacts will all objects correctly. It digs dirt it is on, it can’t walk over boulders but dies to a falling boulder, etc. I ran around and made sure it could pick up all the items, and if it would make barrels and gold visible by being at a distance of 4 or below. To test the moving functionality of Frackman, I made sure that it could not run out of bounds or be too close to Boulders. I also made sure that all of its private data members, such as number of squirts, gold, and water were incremented correctly whenever it ran over an object. The Frackman class also had to be able to shoot squirts, and all of its buttons, such as tab, z, etc were working correctly. I initialized the Frackman with a lot of stuff in its inventory so I could repeatedly test the functionality of all of the objects in its inventory as well as the interactions between the Frackman and all the other objects in the game. For example, the gold nugget could only be grabbed when it is initialized in the dirt. Once a Frackman drops a gold nugget, it can no longer interact or pick up the gold nugget. I tested these thoroughly by running the game a lot and testing each edge case. I tested to see if a Frackman could get annoyed by running a game with one Protester close and many Protesters close to see if it got annoyed repeatedly at the right time.

**Boulder**

I tested the functionality of Boulder by making sure that it blocks Frackman and Protesters from running over it. I also repeatedly dug under the Boulder to make sure it waited for the right amount of ticks before falling. To test the functionality of the falling Boulder, I lined up 5 Protesters below a Boulder and then put the Frackman right below all the Boulders and the Protesters. The Boulder needs to run over the Protesters and kill them, but the Boulder itself should keep falling until it hits another Boulder, dirt, or the Frackman. This specific interaction between Boulders and Protesters was tested repeatedly by running the game a lot and specifically constructing Protesters with coordinates right under a specifically placed Boulder in coordinates right above the Protesters. I also made sure that the Boulder class worked by testing the path finding algorithm to make sure that the map would avoid dirt and Boulders.

**Squirt**

The Squirt class was tested thoroughly by first checking the edge cases. I walked the Frackman over to the very edge of the game, and faced the edge, and pressed Squirt. At first, the Squirts would still show up even though the Squirt image itself was out of bounds. I checked the executable game and noticed that although the Squirt count decrements, the image should not be showing up. So with this testing, I was able to fix my code so that no Squirts show up unless there is actually room for it to show up. Also, I had to check for bounds when the Squirt is moving in its path, and it has to check before it moves every time. To check whether or not the Squirt was working properly, I would start off at 0 units away from the border, and try to squirt, and 1 unit away from broder, and try to shoot, etc. until I was 4 units away from boulder. Only when I was 4 units away from boulder should a Squirt object be made. However, even though it is made, it should stop in its path because its next move would make it go out of bounds. So the Squirt should die right after it is made. I did this testing for all of the edges and for dirt as well. Not even one dirt object can be within the 4x4 box of a squirt or its future path. I checked its interaction with Protesters to make sure the Squirt object died when it contacted a Protester. Lastly, I checked to make sure the Squirt traveled a max distance of 4 before dying, unless it contacted something before its max travel distance.

**Barrel**

I tested this class by running the Frackman close to a Barrel to make sure it shows up before I grab the Barrel. I made sure that the Barrel image would disappear once it gets grabbed. I also tested to make sure all the Barrels were placed at a distance far enough from other barrels and other objects by initiating a map with 30 Barrels. The Barrel class was generally easy to test and as long as it showed up when it was supposed to and disappear when it was supposed to, I knew that the class was working the way it was supposed to.

**Gold Nugget**

I tested this class similarly to the testing of the Barrel class. However, there is a small difference between the Nugget’s interactions with the Frackman compared to the Barrel’s interaction with the Frackman. The Gold Nugget can be grabbed only when it is in its non permanent state. In other words, a Frackman can grab and reveal nuggets when they are first constructed and when they start off as invisible, lying somewhere in the Dirt. Once it grabs it, the gold nugget image should disappear. However, if I press tab while controlling the Frackman, the same gold nugget should reappear again in its non permanent state. That means that a Frackman should not be able to grab the Nugget; it will only be able to be grabbed by Protesters at that point. Furthermore, I tested the lifetime of the Nugget by waiting the right amount of ticks to make sure that the Nugget in its non permanent state disappeared after the amount of ticks described in the spec.

**Sonar Kit**

I tested the Sonar Kit by making sure that it appeared at the right location. Sonar Kits start off as visible, and when it is close to a Frackman, it disappears and increases the sonar count in Frackman’s inventory. This is easily tested, as I just run over the sonar kits to make sure that everything increments the way it is supposed to. After that, I just call a revealCloseObjects function located in StudentWorld whenever the ‘z’ or ‘Z’ key is pressed. After the key is pressed, I decrement the private data member in Frackman that represents the number of Sonar Kits left in his inventory.

**Water Pool**

I tested the Water Pool by making sure that it appears only in spaces not occupied by Dirt. I hard coded in a statement that just printed a lot of Water Pools to make sure it populated only spaces without any dirt in its 4x4 box. I also tested the class to make sure that the number of squirts in Frackman’s inventory increases by 5 when a water pool is grabbed. Lastly, I finished testing water pools to make sure it disappeared either when Frackman grabbed it or when its tickLife runs out, as defined in the spec.

**Protester**

I tested the Protester class by making sure that each of the actions in its doSomething specified in the spec worked the way it was supposed to. I slowly tested each condition by placing Frackman in the place that would trigger the Protesters and cause them to do something specific. For example, I would test the Line of Sight condition by continually walking in and out of the Regular Protester’s view. If I was exactly in line of sight, not blocked by Boulders or dirt, then the Protester would stop what it was doing and face the Frackman and start walking towards the Frackman. If I leave the line of sight after this, the Frackman is allowed to do two things. It will always set a new numSquaresToMove number after Frackman leaves its sight, so sometimes it will walk the opposite direction, and sometimes it would continue to walk in its current direction. Also, I tested the turn function by digging a path perpendicular to the path of the Protester. If the Protester can turn, then it will turn and take the path. If many paths are available, I tested it by putting it at a crossroad to make sure that it would pick a random viable direction. By constructing a Protester in a crossroad, and running the game a lot, I can test to see if it chooses a new path every time. I tested the yell function by walking close to the Protester. To match the executable, I would overlap with the Protester and sometimes be slightly to the left of the Protester and sometimes to the right of the Protester. After the yell counter is over, to match the spec, the Protester would turn to face the Frackman and yell again. It will always turn towards the Frackman first if it is close to the Frackman before it yells. Lastly, I tested the path finding algorithm by creating a complicated path, leading a Protester to the path, and killing it with Squirts. Then, I would see if the Protester would automatically take the best path to (60,60) without running into any items. I would make sure that if it had a crossroad, it would take the shortest path to the end.

**Hardcore Protester**

Most of the functionalities in Hardcore Protester just simply call the functions listed in Protester. So, I only need to test the different functionalities in Hardcore Protester to avoid redundant testing. I know that the Hardcore Protester gets annoyed differently from Regular Protesters. I don’t need to test the Boulder interaction with it because they all die regardless of situation when a Boulder hits the Protesters. However, when a Hardcore Protester picks up a dropped gold nugget, then I know that it will just stare and keep walking. I tested this by continually dropping Gold in front of the Hardcore Protester to make sure that all it did was stare. I also made sure the path finding algorithm for getting close to the Frackman worked. I did this by always placing the Frackman just in the distance threshold and then walking out of the distance threshold to see if the Hardcore Protester would stop chasing the Frackman when it was supposed to. I also would continue to stay within range of being chased and continue to turn and dig myself into the dirt, to see if the Hardcore Protester would make all the right turns and continue to take the best path to get close to the Frackman. A trick to test if the Hardcore Protester takes the most optimal path to the Frackman is to dig a small square path. The Hardcore Protester should start to chase the Frackman in the one direction; however, when the Frackman digs up and connects with its old path, the Hardcore Protester should evaluate whether or not to turn around and chase the Frackman with the new path that was just created as a result of the dirt being cleared, paving way for a faster route to reach the Frackman. I made sure that the Hardcore Protester would always decide to take the best path to reach the Frackman. Once the Frackman was out of the distance threshold and if the Frackman was out of sight, then the Hardcore Protester would act just like a Regular Protester, choosing a number of squares to move in its current direction and changing its direction when the number of squares to move reaches 0 and it picks a new number of squares to move. Overall, the testing of Regular Protesters and the testing of Hardcore Protesters test a number of common methods; testing both of the Protesters thoroughly allows me to know whether or not my classes are working in the right way and are interacting correctly. Since Hardcore Protesters is a derived class of Regular Protesters, the functionality should be similar unless explicitly stated in Hardcore Protesters to be different. This saves time, as no code will be duplicated. It also saves testing time because if I’ve successfully tested the Regular Protesters, I know that all the shared functionalities work so I don’t need to further test the shared functionalities.