Bradley Zhu 304627529

Cs Frackman Report

**A high-level description of each of my public member functions in each of your classes:**

**class Actor: public GraphObject**

**Actor(int imageID, int startX, int startY, Direction dir = right, double size = 1.0, unsigned int depth = 0, bool vis = true);**

This is the constructor for my Actor class. I make it need all the parameters that GraphObject needs, and I pass it right down to GraphObject’s constructor in the initializer list. The one extra parameter is a boolean: vis. It determines whether the object starts off visible or not. I created it with barrels and gold nuggets in mind. It’s default value is true because the vast majority of objects start out visible. I set the object’s visibility to match the bool. Obviously I created this in Actor because it needs a constructor.

**virtual ~Actor();**

This is the destructor for my Actor class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality an actor should do when it is deleted (such as a print statement). This is in Actor because Actor needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

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

This method is the doSomething of Actor. It doesn’t do anything because actors don’t have to do anything. This is in actor because every inherited class should have their own special version of it. In addition, it is a pure virtual function because Actors are abstract classes that should never be instantiated. The reason I made it return a bool rather than be void as in the examples is because I only update the information at the top of the game on the ticks that it has to change rather than every tick. DoSomething will return true if something important happens and that information needs to be updated at the top.

**Direction direc(int x);**

This method takes an int and returns a direction. If given a 0, it will return none, 1: up, 2: right, 3:down, 4:left. It should only ever be given a number in the range [0,5], and that’s a given. It’s a public function in actor because it is just a helper function that I want to be in every subclass. It helps with writing less code, writing it this once rather than every time I need to interpret an integer into a direction. Throughout my entire program I will use the same int to Direction mapping.

**class Dirt: public Actor**

**Dirt(int startX,int startY): Actor(IID\_DIRT, startX, startY, right, .25, 3) {}**

This is the constructor for dirt. All it needs is an x and y coordinate, and everything else is assumed. It calls Actor’s constructor with the correct values, and passes in the X and Y coordinates given by the parameters. All the values that aren’t equal to Actor’s default values are according to the specs. This inherits from Actor rather than from Object later because all it needs is in Actor. It doesn’t need any of the functionality of Object.

**virtual ~Dirt() {}**

This is the destructor for dirt. It doesn’t need to do anything and is virtual for before stated reasons.

**virtual bool doSomething() {**

**return false;**

**}**

This is the doSomething of my Dirt class. It returns false because in no case does the top need to be updated for anything that a Dirt will do. This is virtual because it inherits from the pure virtual doSomething in Actor.

**class Obj: public Actor**

**Obj(StudentWorld\* w, int imageID, int startX, int startY, Direction dir = right, double size = 1.0, unsigned int depth = 0, bool vis = true);**

This is the constructor for my Obj class. This is basically my main class (instead of Actor because dirt doesn’t really need any of this and it doesn’t make sense to inherit from dirt). I make it need all the parameters that Actor needs, and I pass it right down to Actor’s constructor in the initializer list. The default values are what the majority of Objs require. Inside I set two private members that every Obj should have: health and a pointer to the world they are in. Every Object has health, because every object needs to “die” at some point. For some later classes rather than have states I use the health to achieve the same result. The pointer to the StudentWorld is so every inherited subclass will be able to call upon the methods of StudentWorld because only the StudentWorld and not the Obj should know the information about other Objects. Obviously I created this in Obj because it needs a constructor.

**virtual ~Obj();**

This is the destructor for my Obj class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality an Obj should do when it is deleted (such as a print statement). This is in Obj because Obj needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**int getHp() const;**

This method returns the amount of Hp the Object has. It is const because it doesn’t change any private members, only returning one of them. I need to have this for my inherited subclasses to access it. This is not virtual because I should never need to redefine it. This is in Obj because once again, every class I will use will use Hp in some way.

**void setHp(int hp);**

This method takes in an integer and sets the Hp to that integer. Because derived subclasses cannot directly access an inherited class’s private members, I need getters and setters for the private members, in this case only m\_hp. This is void because it never needs to return anything, and it should never really have errors. This is not virtual because I should never have to redefine it.

**virtual bool canPassThrough() const;**

This method returns a boolean concerning whether this object can be passed through. This is const because it doesn’t change any variables. Initially I assigned every class an id, but after learning that this is a bad practice, I changed the id’s into functions that would more or less have the same effect. The default is to return true, because most Obj’s can be passed through. When an object (boulders) can’t, then I’ll redefine it, which is why I made it virtual.

**virtual bool canPickUp() const;**

This method returns a boolean concerning whether this object can be picked up. This is const because it doesn’t change any variables. Initially I assigned every class an id, but after learning that this is a bad practice, I changed the id’s into functions that would more or less have the same effect. The default is to return false, because most Obj’s cannot be picked up. When an object (water or gold or barrels) can, then I’ll redefine it, which is why I made it virtual.

**virtual bool canAnnoy() const;**

This method returns a boolean concerning whether this object can be annoyed. This is const because it doesn’t change any variables. Initially I assigned every class an id, but after learning that this is a bad practice, I changed the id’s into functions that would more or less have the same effect. The default is to return false, because most Obj’s cannot be annoyed. When an object (Frackman or Protesters) can, then I’ll redefine it, which is why I made it virtual.

**virtual bool isHardcore() const;**

This method returns a boolean concerning whether this is “hardcore”. This is const because it doesn’t change any variables. Initially I assigned every class an id, but after learning that this is a bad practice, I changed the id’s into functions that would more or less have the same effect. The default is to return false, because most Obj’s are not “hardcore”. When an object (Hardcore Protesters) is, then I’ll redefine it, which is why I made it virtual.

**void setVis(bool isvis);**

This method sets the object’s visibility to the boolean passed in. This is void because it doesn’t need to return anything and should never run into errors. This is in my Obj class because many derived subclasses will need to use this such as gold or barrels. This is not virtual because it should never have to be redefined.

**virtual bool isAlive() const;**

This method returns a boolean depending on whether the Obj is still alive or not. It accesses m\_hp, because all dead Objs will have an hp of 0 or less, but alive ones will not. I do this rather than use a private boolean. This is in Obj because many subclasses will need to use this, and also StudentWorld will need to call this to make decisions regarding certain objects inside it. This is virtual because some classes (Protesters) aren’t truly dead even when they have no hit points, and I’ll redefine it for them.

**void setCoord(Direction d, int& x, int& y, int r);**

This method takes two references to two ints, a direction, and a distance. I then sets the (x,y) coordinate to the correct one the distance away from given one, in that direction. This is to keep me from having to copy paste the same things in later derived subclasses. It doesn’t do anything if d is none, and doesn’t do any edge checking because other functions will do that. It assumes it will be passed in valid parameters. This is in object because many derived classes will need to update move according to directions, and this helps process where it needs to move to. This is not virtual because it never needs to be changed.

**StudentWorld\* getWorld() const;**

This method returns a pointer to the StudentWorld that the Obj will be in. This is in Obj because every Obj contains a pointer to the StudentWorld it is created in. Then it can call methods in the StudentWorld that will help it perform actions, especially ones that involve interactions with other objects.

**virtual bool damage(int hp);**

This method damages an Obj and returns true if the Obj is dead. This is essentially a setHp that returns a boolean. I can override it for Protester and Frackman because it is virtual and some derived objects do specific things when damaged such as make noises. This is in Obj because my StudentWorld contains a vector of Objects, and I need to call damage on some of them.

**virtual void bribe();**

This method bribes and doesn’t return anything. It is purely here because my StudentWorld contains a vector of Objects, and I need to call bribe on some of them. It doesn’t do anything and is here to be overridden by derived classes.

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

This method is the doSomething of Obj. It doesn’t do anything because Objs don’t have to do anything. This is in Obj because every inherited class should have their own special version of it. In addition, it is a pure virtual function because Objs are abstract classes that should never be instantiated. The reason I made it return a bool rather than be void as in the examples is because I only update the information at the top of the game on the ticks that it has to change rather than every tick. DoSomething will return true if something important happens and that information needs to be updated at the top.

**class Barrel: public Obj**

**Barrel(StudentWorld\* w, int startX, int startY);**

This is the constructor for barrel. All it needs is an x and y coordinate and a pointer to the world it is created in. Everything else is assumed. It calls Obj’s constructor with the correct values, and passes in the X and Y coordinates and the pointer to StudentWorld given by the parameters. All the values that aren’t equal to Obj’s default values are according to the specs. This inherits from Obj rather than from Actor as will everything else because all it need a pointer to a StudentWorld.

**virtual ~Barrel();**

This is the destructor for my Barrel class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality a Barrel should do when it is deleted (such as a print statement). This is in Barrel because Barrel needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual bool canPickUp() const;**

I override canPickUp to return true because Barrels can be picked up. This is for identification purposes in StudentWorld. It is const because it doesn’t change private variables. It is virtual because as before stated, it overrides an existing definition in Obj.

**virtual bool doSomething();**

Like in every doSomething, *if it’s not alive*, I return false. Then *if it’s not visible, but the Frackman is within a Euclidean distance of 4.0* (calling a method in StudentWorld)from it, I set it to be visible, and return false because the score doesn’t need to be changed. If it makes it past that, then *if it’s within a distance of 3.0 from Frackman*, then the Frackman “collects it”. This entails setting the hp to 0 so it “dies”, playing a sound, removing the barrel from the StudentWorld, and returning true because the score needs to be updated. *If none of the above* applied, then return false because nothing needs to be changed.

**class Boulder: public Obj**

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

This is the constructor for boulder. All it needs is an x and y coordinate and a pointer to the world it is created in. Everything else is assumed. It calls Obj’s constructor with the correct values, and passes in the X and Y coordinates and the pointer to StudentWorld given by the parameters. All the values that aren’t equal to Obj’s default values are according to the specs. This inherits from Obj rather than from Actor as will everything else because all it need a pointer to a StudentWorld.

**virtual ~Boulder();**

This is the destructor for my Boulder class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality it should do when it is deleted (such as a print statement). This is in Boulder because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual bool canPassThrough() const;**

This returns false because Boulders can not be passed through. This method is in Boulder because of the above reason (for identification reasons in StudentWorld). This is virtual to override the canPassThrough in Obj.

**virtual bool doSomething();**

This is what Boulder’s doSomething does:

1. If not alive return false as nothing needs to be updated.

2. If the Boulder is currently in the stable state (I use an int), then it must check to see if there is any Dirt in the 4 squares immediately below it (I use a method in StudentWorld). If none of the 4 squares beneath the Boulder have any Dirt, then the Boulder must transition into a waiting state for the next 30 ticks.

3. If the Boulder is in a waiting state and 30 ticks have elapsed, then it must transition into a falling state and play the sound SOUND\_FALLING\_ROCK.

4. If the Boulder is in a falling state, then:

A. It must continue to move downward one square during each tick until it either (a) hits the bottom of the oil field , (b) runs into the top of another Boulder, or (c) it runs into Dirt. If so, the Boulder must set its state to dead.

B. If the Boulder comes within a radius of 3 of any Protester(s) or the FrackMan while falling, it must cause 100 points of annoyance to those actors. The Boulder will continue falling down as described above, even if it annoys one or more Protesters on its way down. However, if the Boulder annoys the FrackMan, the player will instantly lose a life.

This method is in Boulder because every Boulder needs a doSomething method. Also, because Obj defined doSomething as a pure abstract method, every derived subclass needs an implementation of it to be instantiated, which Boulders need to be. It is virtual to override the doSomething in Obj.

**class Water: public Obj**

**Water(StudentWorld\* w, int startX, int startY);**

This is the constructor for Water. All it needs is an x and y coordinate and a pointer to the world it is created in. Everything else is assumed. It calls Obj’s constructor with the correct values, and passes in the X and Y coordinates and the pointer to StudentWorld given by the parameters. All the values that aren’t equal to Obj’s default values are according to the specs. This inherits from Obj rather than from Actor as will everything else because all it need a pointer to a StudentWorld.

**virtual ~Water();**

This is the destructor for my Water class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality it should do when it is deleted (such as a print statement). This is in Water because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual bool doSomething();**

This is what Water’s doSomething does:

1. Return immediately if not alive.

2. Otherwise, if the Water pool is within a radius of 3.0 (<= 3.00 units away) from the FrackMan, then the Water pool will activate, and:

The Water pool must set its state to dead, play the sound effect SOUND\_GOT\_GOODIE, tell the FrackMan object that it just received 5 new squirts of water so it can update its inventory, and increase the player’s score by 100 points.

3. Since the Pool of Water is always in a temporary state, it will check to see if its tick lifetime has elapsed, and if so it must set its state to dead.

This method is in Water because every Water needs a doSomething method. Also, because Obj defined doSomething as a pure abstract method, every derived subclass needs an implementation of it to be instantiated, which Water needs to be. It is virtual to override the doSomething in Obj.

**class Squirt: public Obj**

**Squirt(StudentWorld\* w, int startX, int startY, Direction d);**

This is the constructor for Squirt. All it needs is an x and y coordinate, a Direction and a pointer to the world it is created in. Everything else is assumed. It calls Obj’s constructor with the correct values, and passes in the X and Y coordinates, the Direction, and the pointer to StudentWorld given by the parameters. They need to be constructed in the direction that the Frackman is currently facing, so passing in a Direction is the obvious solution. All the values that aren’t equal to Obj’s default values are according to the specs. This inherits from Obj rather than from Actor as will everything else because all it need a pointer to a StudentWorld.

**virtual ~Squirt();**

This is the destructor for my Squirt class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality it should do when it is deleted (such as a print statement). This is in Squirt because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual bool doSomething();**

This is what Squirt’s doSomething does:

1. If a Squirt is within a radius of 3.0 of one or more Protesters, it will cause 2 points of annoyance to these Protester(s), and then immediately set its state to dead, so it can be removed from the oil field at the end of the tick. It will return true because the text at the top should be changed.

2. If a Squirt has traveled through its full travel distance, then it immediately sets its state to dead, so it can be removed from the oil field at the end of the current tick. It will return false because nothing of importance happened.

3. Otherwise, the Squirt must check to see if it can move one square in its currently facing direction. If this target location is occupied by either Dirt or a Boulder, then the Squirt immediately sets its state to dead so it can be removed from the oil field at the end of the current tick. It will return false because nothing needs to be updated.

4. Otherwise, the Squirt moves one square forward in its currently-facing direction, and then returns false.

This method is in Squirt because every Squirt needs a doSomething method. Also, because Obj defined doSomething as a pure abstract method, every derived subclass needs an implementation of it to be instantiated, which Squirts need to be. It is virtual to override the doSomething in Obj.

**class Gold: public Obj**

**Gold(StudentWorld\* w, int startX, int startY, bool placed);**

This is the constructor for Gold. All it needs is an x and y coordinate, a boolean of whether it was placed by the Frackman or not, and a pointer to the world it is created in. Everything else is assumed. It calls Obj’s constructor with the correct values, and passes in the X and Y coordinates, the bool visible is determined by the bool placed, and the pointer to StudentWorld given by the parameters. Gold’s visibility depends on the circumstances of its instantiation, so I use a bool to determine it. All the values that aren’t equal to Obj’s default values are according to the specs. This inherits from Obj rather than from Actor as will everything else because all it need a pointer to a StudentWorld.

**virtual ~Gold();**

This is the destructor for my Gold class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality it should do when it is deleted (such as a print statement). This is in Gold because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual bool canPickUp() const;**

I override canPickUp to return true because Barrels can be picked up. This is for identification purposes in StudentWorld. It is const because it doesn’t change private variables. It is virtual because as before stated, it overrides an existing definition in Obj.

**virtual bool doSomething();**

This is what Gold’s doSomething does:

1. Returns false immediately if not alive.

2. Otherwise, if the Gold Nugget is not currently visible AND the FrackMan is within a radius of 4.0 of it then it must make itself visible with setVisible() and immediately return.

3. Otherwise, if the Gold Nugget is pickup-able by the FrackMan and it is within a radius of 3.0 from the FrackMan, then it will activate, and:

a. The Gold Nugget must set its state to dead

b. The Gold Nugget must play a SOUND\_GOT\_GOODIE.

c. The Gold Nugget increases the player’s score by 10 points.

d. The Gold Nugget must tell the FrackMan object that it just received a new Nugget so it can update its inventory. Returns true cause top changed

4. Otherwise, if the Gold Nugget is pickup-able by Protesters and it is within a radius of 3.0 (<= 3.00 units away) from a Protester, then the Gold Nugget will activate and set its state to dead, play SOUND\_PROTESTER\_FOUND\_GOLD, tell the Protester object that it just received a new Nugget so it can react appropriately (e.g., be bribed), increase the player’s score by 25 points

5. If the Gold Nugget has a temporary state, then it will check to see if its tick lifetime has elapsed, and if so it must set its state to dead.

This method is in Gold because every Gold needs a doSomething method. Also, because Obj defined doSomething as a pure abstract method, every derived subclass needs an implementation of it to be instantiated, which Gold needs to be. It is virtual to override the doSomething in Obj.

**class Sonar: public Obj**

**Sonar(StudentWorld\* w, int startX, int startY, int hp);**

This is the constructor for Sonar. All it needs is an x and y coordinate, and integer representing hp, and a pointer to the world it is created in. Everything else is assumed. It calls Obj’s constructor with the correct values, and passes in the X and Y coordinates and the pointer to StudentWorld given by the parameters. All the values that aren’t equal to Obj’s default values are according to the specs. The hp int is determined by StudentWorld as it depends on the level and is passed in. The constructor sets the hp equal to this integer, and then reduces it by one every tick. This inherits from Obj rather than from Actor as will everything else because all it need a pointer to a StudentWorld.

**virtual ~Sonar();**

This is the destructor for my Sonar class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality it should do when it is deleted (such as a print statement). This is in Sonar because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual bool doSomething();**

This is what Sonar’s doSomething does:

1. Return false immediately if not alive.

2. Otherwise, if the Sonar Kit is within a radius of 3.0 (<= 3.00 units away) from the FrackMan, then the Sonar Kit will activate, and set its state to dead, play SOUND\_GOT\_GOODIE, tell the FrackMan object that it just received a new Sonar Kit so it can update its inventory, and increases the player’s score by 75 points.

3. Since the Sonar Kit is always in a temporary state, it will check to see if its tick lifetime has elapsed, and if so it must set its state to dead.

This method is in Sonar because every Sonar needs a doSomething method. Also, because Obj defined doSomething as a pure abstract method, every derived subclass needs an implementation of it to be instantiated, which Sonar needs to be. It is virtual to override the doSomething in Obj.

**class Protester: public Obj**

**Protester(StudentWorld \*w, int wait, int iid);**

This is the constructor for Protester. It doesn’t need an x and y coordinate because they all start at 60,60 and I can directly pass that in Obj’s constructor. It needs an integer that represents the wait tick, (It can move one / wait ticks). I pass in an iid because I know hardcore protester will inherit from that, so I need to pass it in. It calls Obj’s constructor with the correct values, including the given pointer to the StudentWorld and the IID. All the values that aren’t equal to Obj’s default values are according to the specs. The int wait is used to determine if the Protester can move or not on a given tick. It is calculated by StudentWorld as it depends on the level and is passed in. This inherits from Obj rather than from Actor as will everything else because all it need a pointer to a StudentWorld.

**virtual ~Protester();**

This is the destructor for my Protester class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality it should do when it is deleted (such as a print statement). This is in Protester because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual bool isAlive() const;**

For protester I have to redefine isAlive. This is in Protester to help my StudentWorld decide when to delete an obj. For Protester it’s not truly dead when its hit points reach 0. It just starts behaving in a different way. It truly dies when It’s hit points reach 0, and it reaches the coordinates (60,60). I redefine it to only return true if the above statement is met. This is virtual because I’m redefining it from Obj, and it helps with readability.

**virtual bool canAnnoy() const;**

I’m redefining canAnnoy in Protester because it shouldn’t return false. It should return true. This is here to help StudentWorld run it’s methods with more information. Otherwise I wouldn’t be able to tell which of my Objs are protesters. This is virtual because I’m redefining it from Obj, and it helps with readability.

**virtual bool damage(int hp);**

This method damages the Protester with the given amount of hit points to deduct. This also updates the wait time (from the game as an example) and plays a sound. This is called by my StudentWorld to damage my Protester if a squirt or boulder hits it. This returns false if the Protester dies, true otherwise. It determines the sound to play on whether the Protester dies or not. This is in Protester because StudentWorld calls this method to damage Protesters. This is virtual because I’m redefining it from Obj, and it helps with readability.

**virtual void bribe();**

This method bribes the protester because it “picked up” gold. It basically plays a sound, sets the hit points to zero, and increases the score. This is called by my StudentWorld to bribe my Protester if it goes over a gold. This doesn’t return anything because nothing needs to be returned. This is in Protester because StudentWorld calls it for Protesters. This is virtual because I’m redefining it from Obj, and it helps with readability, and also because I’ll redefine it for Hardcore Protesters later on.

**virtual bool doSomething();**

This is what Protester’s doSomething does:

1. The Regular Protester must check to see if it is currently alive. If not, then it's doSomething() method must return false immediately.

2. If the Regular Protester is in a “rest state” during the current tick, it must do nothing other than update its resting tick count, and immediately return. I do this by using a protected method. If it returns false I will return false.

3. Otherwise, if the Regular Protester is in a leave-the-oil-field state (its hp is less than or equal to zero but it’s still alive), then call gaveUp and return false as nothing needs to be changed.

4. Otherwise, the Regular Protester will check to see if they are within a distance of 4 units of the FrackMan, AND they are currently facing in the FrackMan’s direction using a method from studentWorld. According to the specs, if both are true and the Regular Protester hasn’t shouted within its last non-resting 15 ticks, then the Regular Protester do it's Shout. However, according to the example game, it sets its wait counter to 45 ticks, and can immediately shout upon exiting it. I’m going with the game. It calls doShout, which returns true if it can shout. If it returns true, then my doSomething returns true as the health changed and I need to update that.

5. Otherwise, if the Regular Protester is in a straight horizontal or vertical line of sight to the FrackMan, is more than 4 units away from the FrackMan, and can move the entire way to the FrackMan with no Dirt or Boulders blocking its path, then the Regular Protester will call see Frackman and return false as nothing in the text will need to be changed.

6. Otherwise, the Regular Protester can’t directly see the FrackMan. As such, it will call regMove and return false as nothing in the top needs to be updated.

This method is in Protester because every Protester needs a doSomething method. Also, because Obj defined doSomething as a pure abstract method, every derived subclass needs an implementation of it to be instantiated, which Sonar needs to be. It is virtual to override the doSomething in Obj, and to let Hardcore Protester override it later.

**Protected functions of Protester:**

I use protected functions because HardCore protester does much of the same thing, and I can reuse this methods. It’s not public because nothing other than inherited classes should use these methods. These are not virtual because they don’t need to be redefined.

**bool canMove();**

If the Regular Protester is in a “rest state” during the current tick, it must do nothing other than update its resting tick count, and immediately return false. Otherwise reset the rest tick and decrement necessary counters and return true.

**void gaveUp();**

The Regular Protester must move one square closer to its exit point using a queue-based maze-searching algorithm in StudentWorld. After moving one step toward their exit point, the method will be done.

**bool doShout();**

If it can shout, this method plays SOUND\_PROTESTER\_YELL, informs the FrackMan that he’s been annoyed for a total of 2 annoyance points, and sets its wait counter to 45. The specs say to update some state variable in a manner that prevents this Regular Protester from shouting again for at least 15 non-resting ticks, but the example game does not behave this way, rather doing what I made my method do. It returns true if it shouted, false otherwise.

**bool seesFrackman();**

This method changes the protester’s direction to face in the direction of the FrackMan, AND then take one step toward him. The Regular Protester will set its squaresToMove counter to zero, forcing it to pick a new direction/distance to move during its next non-resting tick. Then the method will immediately return true.

**void regMove();**

Here is it’s pseudocode:

if it’s squaresToMove counter is zero or below, then it can turn:

make a random array of four directions

Going through my array:

If it's a valid move:

change directions to there

get out of my for loop

Set the squaresToMove counter to a random number between 8 and 60.

Otherwise if the waitTurn counter is below zero:

Call doTurn on with it’s direction and coordinates.

Move one square forwards if we’re able to (nothing blocks us and we are still on map)

Otherwise:

We can’t move so set our squaresToMove counter to zero.

Here is what this method has to do:

regMove decrements its squaresToMove variable by one. If the Regular Protester has finished walking squaresToMove steps in its currently-selected direction then:

a. The Regular Protester will pick a random new direction to move.

b. If the random direction is blocked, then it will continue checking directions until it has picked a direction that is not blocked.

c. The Regular Protester will then change its direction to this new chosen direction.

d. The Regular Protester will pick a new value for squaresToMove in between 8 and 60.

e. The Regular Protester will then jump over the next part

Otherwise if the Regular Protester is sitting at an intersection where it could turn and move at least one square, and it hasn’t made a perpendicular turn in the last 200 non-resting ticks. Then the Regular Protester will:

a. Determine which of the two perpendicular directions are viable.

b. Pick a viable perpendicular direction. If both perpendicular directions are viable, then pick one of the two choices randomly.

c. Set its direction to the selected perpendicular direction.

d. Pick a new value for squaresToMove between 8 and 60.

e. Set a counter back to 200 (ticks to turn)

f. Continue on with the next step.

Finally, the Regular Protester will then attempt to take one step in its currently facing direction. If the Regular Protester is for some reason blocked, it will set squaresToMove to zero, resulting in a new direction being chosen during the Regular Protester’s next non-resting tick (but not the current tick – the Regular Protester must do nothing during the current tick).

**void doTurn(Direction d, int x, int y);**

Given the direction the Protester is currently in and it’s coordinates, it turns in a perpendicular direction if it’s able to. If both perpendicular directions are turnable into, then it randomly chooses one. It is void because it doesn’t need to return anything

**void setWait(int x);**

Sets the private wait variable to the given integer. This is primarily for HardCore Protesters to use as they can’t access this class’s privates.

**class HardcoreProtester: public Protester**

**HardcoreProtester(StudentWorld \*w, int wait);**

This is the constructor for HardcoreProtester. It doesn’t need an x and y coordinate because they all start at 60,60 and I can directly pass that in Obj’s constructor. It needs an integer that represents the wait tick, (It can move one / wait ticks). It calls Protester’s constructor with the correct values, including the given pointer to the StudentWorld and the wait integer. The int wait is used to determine if the HardCoreProtester can move or not on a given tick. It is calculated by StudentWorld as it depends on the level and is passed in. This inherits from Protester rather than from Obj because it is essentially the same thing with a bit of added functionality.

**virtual ~HardcoreProtester();**

This is the destructor for my Protester class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality it should do when it is deleted (such as a print statement). This is in Protester because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual void bribe();**

This method bribes the hardcoreprotester because it “picked up” gold. It basically plays a sound, sets the wait ticker to a given number, and increases the score. This is called by my StudentWorld to bribe my HardcoreProtester if it goes over a gold. This doesn’t return anything because nothing needs to be returned. This is in HardcoreProtester because StudentWorld calls it for HardcoreProtesters. This is virtual because I’m redefining it from Protester because it works slightly differently, and doing so helps with readability.

**virtual bool isHardcore() const;**

This method returns true because hardcore protesters are hardcore. This method is used in StudentWorld to determine which objects are hardcore. It can do certain things to hardcore objects that it can’t do to other things. This function is here to redefine it. This function is virtual because I’m overriding Obj’s function, and having virtual there helps with readability.

**virtual bool doSomething();**

This is what Hardcore Protester’s doSomething does:

1. The Hardcore Protester must check to see if it is currently alive. If not, then it's doSomething() method must return false immediately.

2. If the Hardcore Protester is in a “rest state” during the current tick, it must do nothing other than update its resting tick count, and immediately return. I do this by using a protected method. If it returns false I will return false.

3. Otherwise, if the Hardcore Protester is in a leave-the-oil-field state (its hp is less than or equal to zero but it’s still alive), then call gaveUp and return false as nothing needs to be changed.

4. Otherwise, the Hardcore Protester will check to see if they are within a distance of 4 units of the FrackMan, AND they are currently facing in the FrackMan’s direction using a method from studentWorld. According to the specs, if both are true and the Regular Protester hasn’t shouted within its last non-resting 15 ticks, then the Regular Protester do it's Shout. However, according to the example game, it sets its wait counter to 45 ticks, and can immediately shout upon exiting it. I’m going with the game. It calls doShout, which returns true if it can shout. If it returns true, then my doSomething returns true as the health changed and I need to update that.

5. Otherwise, if the Hardcore Protester is more than 4.0 units away from the FrackMan, the Hardcore Protester but within a value M which is 16 + current level, the call a private function closeToFrack. It basically accesses a queue-based maze-searching algorithm to determine which horizontal/vertical direction to move in to be one square closer to the FrackMan. It returns false as nothing important happened.

6. Otherwise, if the Hardcore Protester is in a straight horizontal or vertical line of sight to the FrackMan, is more than 4 units away from the FrackMan, and can move the entire way to the FrackMan with no Dirt or Boulders blocking its path, then the Hardcore Protester will call see Frackman and return false as nothing in the text will need to be changed.

7. Otherwise, the Hardcore Protester can’t directly see the FrackMan. As such, it will call regMove and return false as nothing in the top needs to be updated.

This method is in HardcoreProtester because every HardcoreProtester needs a doSomething method. It is virtual to override the doSomething in Protester and for better readability.

**class Frackman: public Obj**

**Frackman(StudentWorld\* w);**

This is the constructor for Frackman. All it needs is a pointer to the world it is created in. Everything else is assumed. It calls Obj’s constructor with the correct values, and passes in the pointer to StudentWorld given by the parameters. All the values that aren’t equal to Obj’s default values are according to the specs. This inherits from Obj rather than from Actor as will everything else because all it need a pointer to a StudentWorld. Inside it sets some private members to their initial values.

**virtual ~Frackman();**

This is the destructor for my Frackman class. I don’t do anything inside it because I don’t have any dynamically added objects I need to delete. In addition, there isn’t any functionality it should do when it is deleted (such as a print statement). This is in Frackman because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**int getSquirts() const;**

This is an accessor for a private member that StudentWorld can be used to find how many squirts are in Frackman to put in the text at the top. It is const because nothing is changed inside it. This is in Frackman because the StudentWorld needs to get it from the Frackman object. This is not virtual because nothing will inherit from Frackman.

**void addSquirts();**

This is an accessor for a private member that StudentWorld can use to add squirts to Frackman. This is void because it doesn’t need to return anything. Every time this method is called by StudentWorld it will add 5 squirts to Frackman’s inventory. This is in Frackman because the StudentWorld needs to get it from the Frackman object. This is not virtual because nothing will inherit from Frackman.

**int getGold() const;**

This is an accessor for a private member that StudentWorld can be used to find how many gold Frackman has to put in the text at the top. It is const because nothing is changed inside it. This is in Frackman because the StudentWorld needs to get it from the Frackman object. This is not virtual because nothing will inherit from Frackman.

**void addGold();**

This is an accessor for a private member that StudentWorld can use to add gold to Frackman. This is void because it doesn’t need to return anything. Every time this method is called by StudentWorld it will add 1 gold to Frackman’s inventory. This is in Frackman because the StudentWorld needs to get it from the Frackman object. This is not virtual because nothing will inherit from Frackman.

**int getSonar() const;**

This is an accessor for a private member that StudentWorld can be used to find how many sonars Frackman has to put in the text at the top. It is const because nothing is changed inside it. This is in Frackman because the StudentWorld needs to get it from the Frackman object. This is not virtual because nothing will inherit from Frackman.

**void addSonar();**

This is an accessor for a private member that StudentWorld can use to add sonars to Frackman. This is void because it doesn’t need to return anything. Every time this method is called by StudentWorld it will add 1 sonar to Frackman’s inventory. This is in Frackman because the StudentWorld needs to get it from the Frackman object. This is not virtual because nothing will inherit from Frackman.

**virtual bool doSomething();**

This is what Frackman’s doSomething does:

1. Returns false if not alive.

2. Calls on a method in StudentWorld to delete overlapping dirt objects, and to play SOUND\_DIG if any dirt was removed this tick.

3. If the user pressed a key:

a. If the user presses the Escape key, this allows the user to abort the current level. In this case, the FrackMan object should set itself to completely annoyed.

b. If the user presses the spacebar key, then the FrackMan will fire a Squirt into the oil field, assuming they have sufficient water in their squirt gun to do so. I use a private method to do the following: The FrackMan will then reduce their water count by 1. To fire a Squirt, the player must create and add a new Squirt object into the oil field at a location that is four squares immediately in front of the FrackMan, facing the same direction as the FrackMan. Every time the FrackMan fires a Squirt, it must play the sound SOUND\_PLAYER\_SQUIRT. If the initial location where the Squirt starts is occupied by one or more Dirt objects or is within a radius of 3.0 of a Boulder object, then the FrackMan must still make a squirting noise and will waste 1 unit of water, but the FrackMan must not add any Squirt object into the oil field.

c. If the player wants their FrackMan to turn to face a different direction then it is currently facing, then the FrackMan’s direction should be adjusted to the indicated direction but the FrackMan must not move in that direction.

d. Else, if the FrackMan is already facing in a particular direction and the user hits the arrow key to move in that same direction, AND the FrackMan is allowed to move in that direction, then your code must use the GraphObject’s moveTo() method to adjust the FrackMan’s location by one square in the indicated direction. Valid coordinates for the FrackMan are from x=0,y=0 to x=60,y=60, inclusive. Moreover, the FrackMan cannot occupy a square that is less than or equal to a radius of 3 away from the center of any Boulder.

e. If the player presses the ‘Z’ or ‘z’ keys and the FrackMan has one or more sonar charges remaining in his inventory, then this will cause the FrackMan to use his sonar charge to illuminate the contents of the oil field within a radius of 12 of his location. I use a method to do the following: i. The count of sonar charges held by the FrackMan must be decremented by 1. ii. All hidden game objects that are within a radius of 12 must be made visible via setVisible() and revealed to the player.

f. If the player presses the TAB key and they have one or more units of gold in their inventory, then they must add a Gold Nugget object into the oil field at their current x,y location and reduce their gold count by 1 unit. I use a method to do this. The dropped Gold Nugget must have a temporary state and a lifetime of 100 game ticks, and is only pickup-able by Protesters. Such a dropped Nugget will start out in a visible state.

This method is in Frackman because every Frackman needs a doSomething method. Also, because Obj defined doSomething as a pure abstract method, every derived subclass needs an implementation of it to be instantiated, which Sonar needs to be. It is virtual to override the doSomething in Obj and for better readability.

**class StudentWorld : public GameWorld**

**StudentWorld(std::string assetDir) : GameWorld(assetDir) {}**

This constructor basically just passes the parameter on to the GameWorld constructor in the initializer list. I don’t have anything inside because everything is created in the init function because it needs new versions of everything every level.

**virtual ~StudentWorld();**

In this destructor I delete every dynamically allocated object including my two dimensional array of dirt, my vector of objs, and my frackman. This is in StudentWorld because it needs a destructor. It is a virtual function because all destructors should be virtual when inheritance is involved.

**virtual int init();**

This method is called every time my FrackMan dies and has to restart the level and every time he collects all the barrels and he can advance to the next level. In it I initialize a lot of private members including integers, Frackman, and all sorts of Obj. It figures out how many boulders, gold, and barrels of oil should be on the map, as well as all the different counters for protesters. It randomly puts what it needs to put in into the map calling a method canPlace to make sure the place is fine. It also sets the display for the first time using setDisplay. It returns GWSTATUS\_CONTINUE\_GAME. This is virtual because GameWorld (what this inherits from) has the function too as a pure abstract function, making it an abstract class. Therefore StudentWorld has to implement it, and virtual is to make my code easier to follow.

**int min(int i, int j) const;**

This method is just to help me determine some of the values that are a minimum between two values. It takes the two integers given as parameters and returns the smaller one. If they are equal it’ll return either one of them. I have this as public rather than private because some functions in my Actor.cpp file use it as well. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**int max(int i, int j) const;**

This method is just to help me determine some of the values that are a maximum between two values. It takes the two integers given as parameters and returns the bigger one. If they are equal it’ll return either one of them. I have this as public rather than private because some functions in my Actor.cpp file use it as well. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool onDirt(int x, int y) const;**

This method determines if there is any dirt below the four pixels of the given (x,y) coordinate. It will return true if it is, false otherwise. If given invalid coordinates it will return false as well. This is a public method in StudentWorld because Boulders can use this while falling to see if they can continue to fall. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool onBoulder(int x, int y) const;**

This method determines if there is any boulder pixels below the four pixels of the given (x,y) coordinate. It will return true if it is, false otherwise. This is a public method in StudentWorld because Boulders can use this while falling to see if they can continue to fall. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool clearDirtBoulder(int x, int y) const;**

This method determines if there is any boulder within 3.0 of the given (x,y) coordinate or dirt four pixels in a 4x4 above and to the right of the given (x,y) coordinate. It will return true if it is clear of them, false otherwise. When given invalid coordinates it will also return false. This is a public method in StudentWorld because Squirts and Protesters can use to see if they can move to a given coordinate. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

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

This method removes dirt in a 4x4 box above and to the right of the given (x,y) coordinate. It will return true if any dirt was removed, false otherwise. When given invalid coordinates it will also return false. This is a public method in StudentWorld because Frackman uses it to remove the dirt where it moves and to determine if it needs to play a sound or not. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**void removeBarrel();**

This method just decrements a counter of barrels and increases the score of the game. This is a public method in StudentWorld because Barrels use it to make sure the StudentWorld is aware that a barrel was picked up. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool collFrack(int x, int y, double r) const;**

This method just determines whether a given (x,y) coordinate is within a given Euclidian distance of radius r of a Frackman. It returns true if it is, false if it isn’t. This is a public method in StudentWorld because every Obj that is picked up uses it to make sure the StudentWorld is aware that a it was picked up. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool annoyP(int x, int y, double r, int hp, int rpoints, int hpoints);**

This method annoys protesters for a given hp within a Euclidian distance r of the given (x,y) coordinate. It will return true if any Protesters were annoyed, false otherwise. It will also increase the score by rpoints if a Protester dies and by hpoints if a Hardcore Protester dies. This is a public method in StudentWorld because boulders and squirts use it to annoy Protesters at its given coordinates. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool annoyF(int x, int y, int r, int points) const;**

This method annoys the Frackman if it’s within a Euclidian distance r of the given (x,y) coordinate. It will return true if the Frackman was annoyed, false otherwise. This is a public method in StudentWorld because boulders use it to annoy the Frackman at its given coordinates. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool bribeP(int x, int y) const;**

This method bribes protesters within a Euclidian distance 3.0 of the given (x,y) coordinate. It will return true if any Protesters were bribed, false otherwise. This is a public method in StudentWorld because gold uses it to bribe Protesters at its given coordinates. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool canMove(int x, int y) const;**

This method checks to see if there is a boulder in the way (radius 3.0). It will return true there isn’t, false otherwise. This is a public method in StudentWorld because Frackman uses it to see if it can move. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**bool canShout(int x, int y, Actor::Direction d);**

This method checks to see if there if there is a Frackman within a 4.0 radius and in the given direction of the (x,y) coordinate. It will return true there isn’t, false otherwise. This is a public method in StudentWorld because Protester uses it to see if it can shout at the Frackman. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**int straightFrack(int x, int y, int r) const;**

This method checks to see if there if there is a Frackman within eye distance of a given Frackman. The integer r is a limit, so if it is 0, it can be 0 blocks away and if it's 4, it has to be at least 4 blocks away. It returns an int representing the direction. It returns a -1 for can’t see the Frackman, 0 for on Frackman, 1 for up, 2 for right, 3 for down, and 4 for left. This is a public method in StudentWorld because Protester uses it to see if the Frackman is in a straight line from it. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**int isNearFrack(int &x, int &y) const;**

This method checks to see if there if it is near a Frackman (given manhattan distance). It also modifies the given x and y integers to a coordinate that’s 1 closer to Frackman. It returns a 0 for not near Frackman, 1 for below, 2 for to the left, 3 for above, and 4 for to the right. This is a public method in StudentWorld because Hardcore Protesters use it to go towards the Protester if they are within the given manhattan distance. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**int annoyedProtester(int &x, int &y);**

This method helps a fully annoyed Protester find his way to the exit (60,60). It also modifies the given x and y integers to the best coordinate that’s 1 closer to the exit using a queue-based maze searching algorithm. It returns a 0 if it can’t move, 1 to move up, 2 to move right, 3 to move down, and 4 to move left. This is a public method in StudentWorld because Hardcore Protesters and Protesters use it to go towards the exit if they are annoyed. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**void squirt(int x, int y);**

This method creates a new squirt object with the given x and y coordinate and the Frackman’s direction. It then puts the newly created squirt object into the vector of objs. This is a public method in StudentWorld because the Frackman calls it to create a new Squirt in the correct location in StudentWorld. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**void addSquirts() const;**

This method can be called by any of the Objects inside Studentworld to add a squirt to Frackman. It calls Frackman’s method to do all the internal changes that need to be made. This is void because it doesn’t need to return anything. Every time this method is called by any Object, 5 squirts will be added to Frackman’s inventory. This is in StudentWorld to let other objects interact with the Frackman. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**void addGold() const;**

This method can be called by any of the Objects inside Studentworld to add a gold to Frackman. It calls Frackman’s method to do all the internal changes that need to be made. This is void because it doesn’t need to return anything. Every time this method is called by any Object, 1 gold will be added to Frackman’s inventory. This is in StudentWorld to let other objects interact with the Frackman. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**void addSonar() const;**

This method can be called by any of the Objects inside Studentworld to add a sonar to Frackman. It calls Frackman’s method to do all the internal changes that need to be made. This is void because it doesn’t need to return anything. Every time this method is called by any Object, 1 sonar will be added to Frackman’s inventory. This is in StudentWorld to let other objects interact with the Frackman. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**void placeGold(int x, int y);**

This method creates a new gold object with the given x and y coordinate and makes the value of places true. It then puts the newly created gold object into the vector of objs. This is a public method in StudentWorld because the Frackman calls it to create a new Gold in the correct location in StudentWorld. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**void reveal(int x, int y);**

This method reveals all nearby hidden objects in a Euclidian distance of 12.0 from the given (x,y) coordinate. This is a public method in StudentWorld because Sonars use it to interact with other Objects and change their visibility within the StudentWorld. It is not virtual because nothing will inherit from StudentWorld and there is no need to make it virtual because it will never change.

**virtual int move();**

My move() method does the following activities:

1. First increments all my counters.

2. Builds the maps for annoyed Protesters and for Hardcore Protesters who are near the Frackman.

3. Creates Protester if able to. I’m doing this before asking every Object to do something to have Protesters work on the tick they are created.

4. Asks all of the actors that are currently active in the game world to do something

A. If an actor does something that causes the FrackMan to give up, then the move() method should immediately return GWSTATUS\_PLAYER\_DIED.

B. If the FrackMan collects all of the Barrels of oil on the level (completing the current level), then the move() method should immediately play a sound of SOUND\_FINISHED\_LEVEL and then return a value of GWSTATUS\_FINISHED\_LEVEL.

5. Deletes any actors that need to be removed from the game during this tick and removes them from my STL container that tracks them.

6. If something important happened and the text at the top needs to be changed, then change it

7. Put random objects into the game if able to and at random locations.

**virtual void cleanUp();**

In this method, I delete every dynamically allocated object including my two dimensional array of dirt, my vector of objs, and my frackman. This is in StudentWorld because it needs to be able to clean up all dynamically allocated objects between levels, or the memory used will just keep building up. It is a virtual function because all destructors should be virtual when inheritance is involved. This is virtual because GameWorld (what this inherits from) has the function too as a pure abstract function, making it an abstract class. Therefore StudentWorld has to implement it, and virtual is to make my code easier to follow.

**A list of all functionality that you failed to finish as well as known bugs:**

I don’t know of any bugs, and I finished all functionality in the specs.

**A list of other design decisions and assumptions you made:**

When a protester shouts, the specs say to keep the Protester from shouting again in the next 15 non resting ticks. However when playing the game, I realized that the way the game does it is to make the Protester freeze for approximately 45 ticks (I counted). Immediately after regaining the ability to move, it can shout again. Therefore I set my Protester’s wait counter to 45, rather than use a variable that checks how many non resting ticks it has been since the last shout.

The specs were ambiguous about falling boulders falling on top of other boulders. The majority of the game uses Euclidian distances, but this time it doesn’t specify. I decided to make boulders die as soon as their pixels overlay because I believe that would make the most sense.

The specs said to make Protesters spawn in the first tick of every game or restart. The example didn’t do this. I went with the spec.

**A description of how you tested each of your classes:**

**Dirt:**

I made sure they were initialized in the correct places in the grid. I made sure they were being deleted as Frackman ran over them. I made sure that they were deleted when boulders are overlaid on top of them. I made sure certain objects could be hidden in them, and that they were deleted in my cleanup and destructor.

**Barrel:**

I made sure my barrels could be collected. I made sure they were invisible, and that they would become visible when the Frackman was within a radius of 4.0 from them. I made sure they worked with sonar, and that collecting them would increase the score. I made sure the correct number of barrels generated for every level (random generation) and that they could be collected properly. I made sure they didn’t interact weirdly with Protesters and that the correct sound would output when I collect them. I made sure the barrels also worked with level restarts, that they would respawn correctly and randomly.

**Boulder:**

First I made sure they generated properly. They can’t be within a certain radius of each other and the dirt behind them should be destroyed. They also can’t be below y = 20. They should start in a stable state. When the dirt beneath them disappears, they should wait 30 ticks, then start falling. I made sure they would wait the correct amount of ticks and fall one coordinate at a time. They should stop falling and disappear when they either have to run into dirt or a boulder. I made sure that worked properly.

I made sure they interacted with Frackman and Protesters properly. They should run through Protesters and kill them (producing soudns and increasing my score). When it hits a Frackman the game should end because it basically one hits the Frackman. Lastly I made sure that the Frackman would dig around it properly with Euclidean distance. For a while I had trouble figuring out the rules before I realized it was Euclidean.

**Water:**

I made sure water spawned correctly. Every level it should spawn less. It has to spawn in a random location that’s clear of dirt and boulders. When spawning it should spawn with a hp that is according to the level. I tested it by observing the spawn rates over different levels.

I also tested it by collecting it as Frackman. It had to increase my squirts by 5. Also it had to disappear correctly, both after beign collected and after its hitpoints run out. I made sure it increased my score correctly and made the right sounds.

**Squirt:**

I made sure that the Frackman creates Squirts correcly (in the same direction and 4 blocks in front of it). I made sure it lasted the same amount of ticks as in the example. After that it had to disappear. I made sure when they spawned they decreased the amounf of squirts in Frackman’s invetory accordingly.

I made sure the squirt would not spawn if I tried to do it in a place where it can’t spawn with a cout statement. The sound still had to play though. I made sure they interacted with Protesters correctly. 3 Squirts can kill a normal one, and 10 kill a hardcore one. I made sure the Protesters all had to wait after being hit and that a sound would come out. I made sure when they died my score got increased correctly.

The movement was straight fowards. It moved one block a tick in the direction that it was instantiated in. If running into dirt or a boulder it had to disappear.

**Gold:**

I made sure the correct number spawned every level in random locations. I made sure they all spawned hidden and would become visible if I got nearby. I made sure that the Frackman could collect them, the sound would play, the score would increase, and the amount of gold inventory would increase too.

To test the dropping capabilities I made sure that the Frackman could drop them. I made sure it could only do so when it had gold, and that after beign dropped it would disappear after the specified amount of ticks. I made sure regular protesters interacted with them correctly as well as hardcore protesters.

**Sonar:**

I made sure they randomly spawned in the top left corner correctly. I made sure after spawning they would last for the correct amount of ticks. I made sure Frackman could collect them and that the sound would play, the score would increase, and the amount of sonar invetory would increase. I made sure Frackman started with one already in its inventory.

I made sure that the Frackman could use them correctly. I made sure hidden objects nearby turned visible. I made sure that the Euclidean distance worked correctly. They became less frequent the higher level I was.

**Protester:**

I made sure one always spawned in the very first tick of the game (which the example didn’t do). I made sure that after being killed they would go back to the start correctly. I made sure they couldn’t be in the same coordinates as dirt or boulders.

In addition I extensively tested their rules for movement that I’ve listed already. I made sure that they would go towards the Frackman if its in the line of sight, choose a random direction if at a wall, etc. I made sure they collected bribes correctly, that they died when hitting boulders, and that they lost hp when hit with squirts. I made sure after dying and reaching the top right corner they disappeared. I made sure they only moved the specified frequency of ticks. I made sure all the sound worked and the score increases worked.

**Hardcore Protester:**

I made sure that the probablity thing worked correctly between Protesters and Hardcore Protesters. I made sure that after being killed they would go back to the start correctly. I made sure they couldn’t be in the same coordinates as dirt or boulders.

In addition I extensively tested their rules for movement that I’ve listed already. I made sure that they would go towards the Frackman if its in the line of sight or within a certain amount of blocks, choose a random direction if at a wall, etc. I made sure they collected bribes correctly, that they died when hitting boulders, and that they lost hp when hit with squirts. I made sure after dying and reaching the top right corner they disappeared. I made sure they only moved the specified frequency of ticks. I made sure they didn’t try to leave the field when bribed and that all the sound and score increases worked.

**Frackman:**

I made sure all the direction keys worked and they didn’t make him run into boulders. I made sure that dirt disappeared as he dug. I made sure tab and space and escape worked as intended. I made sure his invetory updated correctly whether gaining stuff or losing/using it. I made sure it could interact with other objects correctly such as gaining water, shooting squirts, dying to boudlers, digging dirt, collecting gold or sonar or water pools, placing gold down, being annoyed by protesters in the viscinity, etc. I made sure it had to stay within the map and that it would die signalling game over correctly when annoyed sufficiently.

**StudentWorld:**

I made sure the game started and ended correctly. I made sure all my dynamically allocated objects were being deleted when necessary. I made sure every object was created correctly and put into the vector or 2d array of dirt. I made sure the game ended right when all barrels of oil were collected and that it would advance to the next level. I made sure when the Frackman died the level would restart. I made sure that Protesters would be able to make it back to the top right corner when annoyed to 0 hp. I tested random collisions such as squirt with Protester or boulder with Frackman. I made sure the score worked as intended, and all the sounds played. I made sure the text at the top updated and displayed correctly. I made sure all the numbers were instantiated correctly that depended on the level in the init. I made sure all the methods in it that needed to could be accessed by the objects that used them to accomplish various tasks.