Jason Song

805532320

**High-Level Descriptions of Public Member Functions**

**Actor Class:**

* The *Actor* constructor takes as parameters an image ID, a starting x-coordinate, a starting y-coordinate, a *StudentWorld* pointer, a *Direction* (defaulted to *right*), an image size (defaulted to 1.0), and an image depth (defaulted to 0). All classes directly derived from class *Actor* must call on the *Actor* constructor in their constructors’ initializer lists.
* A virtual destructor is provided. It is empty because the *Actor* constructor does not perform any dynamic memory allocation. Moreover, the virtual *Actor* destructor is a public member function because it is called after derived classes’ destructor implementations are run.
* *Actor::setAlive(bool lifeState)* is a public member function of class *Actor* because every derived class (apart from *Earth*) must be able to set its objects to a “dead” state. Similarly, *Actor::getAlive()* is public because every game actor can be dead or alive.

The *StudentWorld* class needs to call on those two functions when processing its game objects (ex. determining whether the TunnelMan is still alive in *StudentWorld::move()* ).

* I made *Actor::doSomething()* a pure virtual function because an *Actor* object will never need to be created, which makes a function implementation redundant. The *Actor* class can just be an abstract base class. Additionally, all Actors in the game must have a *doSomething()* method (called in *StudentWorld::move()* ), and each type of actor overrides the *Actor* class’s version.
* An *Actor::getWorld()* public member function with return type *StudentWorld\** is included so that the *Actor* class (and its derived classes) can call on *StudentWorld* to manipulate the former’s objects and increase/decrease the player’s lives.
* Because I was not permitted to use *GraphObject::isVisible()*, I added a private boolean member variable called *visible* to denote whether a game actor is visible. So, I included a getter function *Actor::getVisible()* and a mutator function *Actor::setVisible(bool val)* so that derived classes and the *StudentWorld* class can adjust or refer to an actor’s visibility. For example, *StudentWorld::earthInvisible(int col, int row)* uses *Actor::setVisible(bool val)* to make an *Earth* object at a particular location invisible.
* I created a virtual public identifier function *Actor::isInitiallyDistributedGoodie()* such that only objects randomly distributed in *StudentWorld::init()* (*Barrels, GoldNuggets* and *Boulders*) return true.
* The virtual public identifier function *Actor::isBoulder()* returns true only for class *Boulder*. I made the function public because the *StudentWorld* class’s boulder-searching functions call on it.
* The virtual public identifier function *Actor::isAnnoyable()* returns true only for class *TunnelMan* and *Protestor* (and its derived classes). I made the function public because the *StudentWorld* class’s protestor-searching functions call on it.

**Earth Class:**

* The *Earth* constructor takes a starting x-coordinate, a starting y-coordinate, and a *StudentWorld* pointer as parameters. All other parameters taken in by the *Actor* constructor are provided arguments as detailed in the project spec.
* A virtual destructor is provided. It is empty because the *Earth* constructor does not perform any dynamic memory allocation. The destructor is virtual because *Earth* is a derived class of *Actor*. (The *virtual* keyword is optional here, though.)
* A virtual *Earth::doSomething()* function is provided to override the pure virtual function *Actor::doSomething()*. *Earth::doSomething()* is private because it is not called by any other class. To prevent *Earth* from being an abstract base class, an empty function implementation is empty because, needless to say, *Earth* objects do not do anything.

**Person Class:**

* The *Person* constructor takes the same parameters as those of the *Actor* constructor, in addition to a *points* integer parameter that stores how many hit points a *Person* (*TunnelMan* or *Protestor*) has. All classes directly derived from class *Person* must call on the *Person* constructor in their constructors’ initializer lists.
* As previously mentioned, the public virtual member function *Person::isAnnoyable()* always returns true.
* The virtual *Person* destructor has an empty implementation because the *Person* constructor does not perform any dynamic memory allocation.
* The *Person::getHitPoints()* and *Person::setHitPoints(int val)* public member functions are needed to access and mutate, respectively, the *hitPoints* private member variable of a *Person* object. These functions are made public because a few of the *StudentWorld* class’s functions need to use them when manipulating its game objects.

**TunnelMan Class:**

* The *TunnelMan* constructor only takes a *StudentWorld* pointer as a parameter. All other parameters taken in by the *Person* constructor are provided arguments as detailed in the project spec.
* The virtual *TunnelMan* destructor’s implementation is empty because the *TunnelMan* constructor does not perform any dynamic memory allocation.
* A public virtual *TunnelMan::doSomething()* function is provided to override the pure virtual function *Actor::doSomething()*. *TunnelMan::doSomething()* is public because it must be called in *StudentWorld::move()*.
* *TunnelMan::changeGoldNuggets(int val)* and *TunnelMan::getGold()* mutate and access, respectively, the number of *Gold Nuggets* held by the *TunnelMan*. These functions are made public because a few of the *StudentWorld* class’s functions need to use them.
* *TunnelMan::changeSquirts(int val)* and *TunnelMan::getSquirts()* mutate and access, respectively, the number of *Squirts* held by the *TunnelMan*. These functions are made public because a few of the *StudentWorld* class’s functions need to use them.
* *TunnelMan::changeSonarCharges(int val)* and *TunnelMan::getSonar()* mutate and access, respectively, the number of *SonarKits* held by the *TunnelMan*. These functions are made public because a few of the *StudentWorld* class’s functions need to use them.

**Protestor Class:**

* The *Protestor* constructor takes an image ID, a *StudentWorld* pointer, and a *Direction*.
* The virtual *Protestor* destructor’s implementation is empty because the *Protestor* constructor does not perform any dynamic memory allocation.
* *Protestor::annoyProtestor(int val, bool boulderBonked, boolSquirted)* decrements a *Protestor’s* hit points depending on whether it was bonked by a boulder or squirted on. This function was made public because a few of the *StudentWorld* class’s functions need to use it to damage a *Protestor*.
* *Protestor::bribedProtestor()* is called by the *StudentWorld* class when a *Protestor* picks up a *Golden Nugget* dropped by the *TunnelMan*. The function increases the player’s score, makes a *Regular Protestor* leave the oil field, and forces a *Hardcore Protestor* to pause in its tracks for a few ticks.

**RegularProtestor Class:**

* The *RegularProtestor* constructor only takes a *StudentWorld* pointer. All other private member variables are initialized by the *Protestor* constructor in the initializer list.
* The virtual *RegularProtestor* destructor’s implementation is empty because the *RegularProtestor* constructor does not perform any dynamic memory allocation.
* The public identifier function *RegularProtestor::isRegularProtestor()* returns true.
* The *RegularProtestor* class’s virtual *doSomething()* method just calls on the *Protestor* class’s protected *doSomething()* method.

**Hardcore Protestor Class:**

* The *HardcoreProtestor* constructor only takes a *StudentWorld* pointer. All other private member variables are initialized by the *Protestor* constructor in the initializer list.
* The virtual *HardcoreProtestor* destructor’s implementation is empty because the *HardcoreProtestor* constructor does not perform any dynamic memory allocation.
* The public identifier function *HardcoreProtestorr::isHardcoreProtestor()* returns true.
* The *HardcoreProtestor* class’s virtual *doSomething()* method just calls on the *Protestor* class’s protected *doSomething()* method.

**Goodies Class:**

* The *Goodies* constructor takes an image ID, a starting X-coordinate, a starting Y-coordinate, a *StudentWorld* pointer, a *Direction*, an image size, and an image depth parameter.
* The *Goodies::setGoodiesTickLife(int val)* and *Goodies::getGoodiesTickLife()* public member functions mutate and access the tick lifetime of a *Goodie* object.
* The virtual *Goodies* destructor’s implementation is empty because the *Goodies* constructor does not perform any dynamic memory allocation.

**Barrel Class:**

* The *Barrel* constructor takes a starting X-coordinate, a starting Y-coordinate, and a *StudentWorld* pointer.
* The class’s virtual *doSomething()* method overrides the *Actor* base class’s version.
* As previously mentioned, *Barrel:isInitiallyDistributedGoodie()* always returns true.
* The virtual *Barrel* destructor’s implementation is empty because the *Barrel* constructor does not perform any dynamic memory allocation.

**SonarKit Class:**

* The *SonarKit* constructor takes a starting X-coordinate, a starting Y-coordinate, and a *StudentWorld* pointer.
* The class’s virtual *doSomething()* method overrides the *Actor* base class’s version.
* The virtual *SonarKit* destructor’s implementation is empty because the *SonarKit* constructor does not perform any dynamic memory allocation.

**GoldNugget Class:**

* The *GoldNugget* constructor takes a starting X-coordinate, a starting Y-coordinate, a *StudentWorld* pointer, a boolean parameter which indicates whether or not the *Gold Nugget* can be picked up by the *TunnelMan*, and another boolean which indicates whether the *Gold Nugget* is temporary.
* The class’s virtual *doSomething()* method overrides the *Actor* base class’s version.
* The virtual *GoldNugget* destructor’s implementation is empty because the *GoldNugget* constructor does not perform any dynamic memory allocation.

**Water Class:**

* The *Water* constructor takes a starting X-coordinate, a starting Y-coordinate, and a *StudentWorld* pointer.
* The class’s virtual *doSomething()* method overrides the *Actor* base class’s version.
* The virtual *Water* destructor’s implementation is empty because the *Water* constructor does not perform any dynamic memory allocation.

**Boulder Class:**

* The *Boulder* constructor takes a starting X-coordinate, a starting Y-coordinate, and a *StudentWorld* pointer. *Boulder* being a derived class of *Actor,* all other parameters taken in by the *Actor* constructor are provided arguments as detailed in the project spec.
* The class’s virtual *doSomething()* method overrides the *Actor* base class’s pure virtual version.
* As previously mentioned, *Boulder:isInitiallyDistributedGoodie()* always returns true. Please note that *Boulder* is not a derived class of *Goodie*. *Boulders* are just initially randomly-distributed by *StudentWorld::init()*.
* The virtual *Boulder* destructor’s implementation is empty because the *Boulder* constructor does not perform any dynamic memory allocation.

**Squirt Class:**

* The *Squirt* constructor takes a starting X-coordinate, a starting Y-coordinate, a *StudentWorld* pointer, and a *Direction*. *Squirt* being a derived class of *Actor,* all other parameters taken in by the *Actor* constructor are provided arguments as detailed in the project spec.
* The class’s virtual *doSomething()* method overrides the *Actor* base class’s pure virtual version.
* The virtual *Squirt* destructor’s implementation is empty because the *Squirt* constructor does not perform any dynamic memory allocation.

**StudentWorld Class:**

* **StudentWorld(std::string assetDir);**

Constructs a StudentWorld object, setting the private *TunnelMan* pointer variable to *nullptr.*

* **virtual int init(); //NEVER call**

Initializes gameplay, randomly distributing *Barrels, Boulders,* and *Gold Nuggets*.

* **virtual int move(); //NEVER call**

Ask each living game object to move.

* **virtual void cleanUp(); //NEVER call**

**Destroy all remaining game objects.**

* **~StudentWorld();**

Destroy StudentWorld object.

* **void earthInvisible(int col, int row);**

**Set Earth object to be invisible at location (col, row).**

* **bool getEarthVisibility(int col, int row);**

**Retrieve visibility of Earth object at location (col, row).**

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

**Return false if there are no Earth objects in 4x4 square with lower-left corner at (x,y).**

* **double computeDistanceToTM(Actor\* actor);**

**Compute distance of actor to TunnelMan.**

* **void subtractBarrel();**

**Decrement level’s Barrel count by one when players picks up Barrel**

* **void addToList(Actor\* actor);**

**Add Actor to private member STL list of all game actors (other than TunnelMan and Earth objects).**

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

**Returns true if there is no Boulder within radius of 3.0 from (x,y).**

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

**Makes visible all hidden game objects that are within radius of 12 from (x,y).**

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

**Returns true if there is a Boulder at (x,y).**

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

**Returns true if there is a Protestor at (x,y).**

* **void squirtAssistant(Squirt\* s);**

**If a Squirt is within a radius of 3.0 of one or more Protesters (up to and including a distance of 3.0 squares away), 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.**

* **Protestor\* goldNuggetProtesterFinder(GoldNugget\* g);**

**Returns pointer to a Protestor (if any) within radius of 3 from Gold Nugget situated at (x,y)**

* **void boulderBonker(Boulder \*b);**

**Bonks any Protestor within radius of 3 of Boulder.**

* **void annoyPerson(Actor \*a, int val);**

**Annoy a Person by int “val”.**

* **void annoyTunnelMan(int val);**

Decrease TunnelMan’s hit point amount by “val”.

* **void changeSonarAmount(int val);**

Increment TunnelMan’s sonar kit amount by “val”.

* **void changeGoldAmount(int val);**

Increment TunnelMan’s gold amount by “val”.

* **void changeSquirtAmount(int val);**

Increment TunnelMan’s water squirt amount by “val”.

* **int getTunnelManHitPoints();**

Get the TunnelMan’s number of remaining hit points.

* **int getTunnelManGoldAmount();**

Get the TunnelMan’s number of Gold Nuggets.

* **int getTunnelManSonarAmount();**

Get the TunnelMan’s number of SonarKits.

* **int getTunnelManSquirtAmount();**

Get the TunnelMan’s number of squirts.

* **int getTunnelManXCoord();**

Get the TunnelMan’s x-coordinate.

* **int getTunnelManYCoord();**

Get the TunnelMan’s y-coordinate.

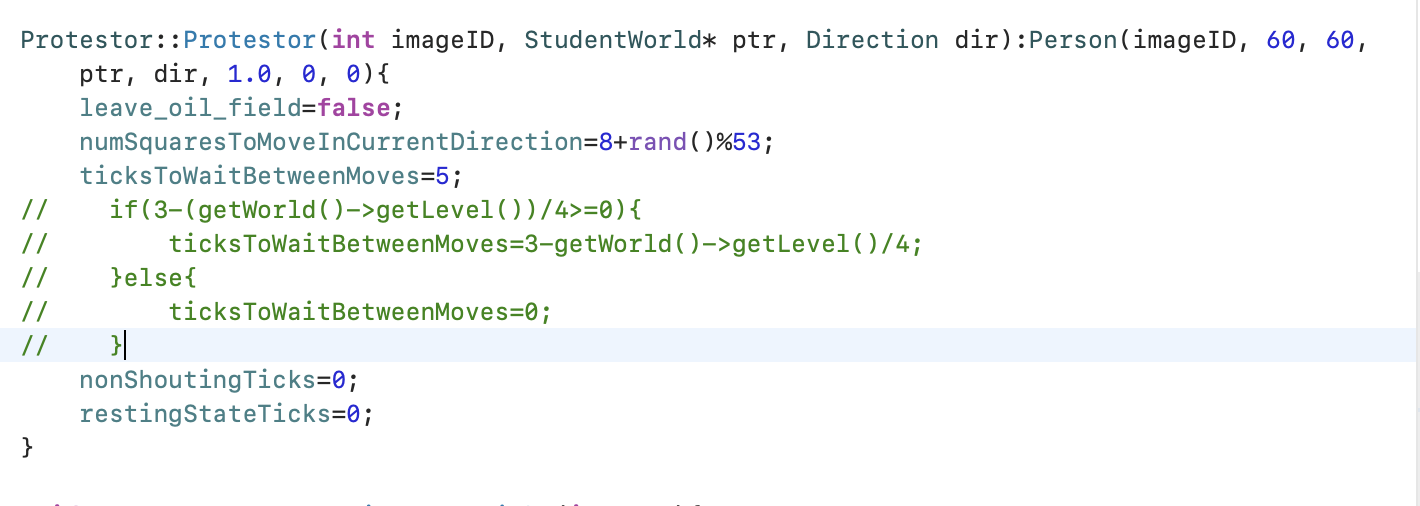
* **void killTunnelMan();**

Kills the TunnelMan, reducing its hit points below or equal to 0. One life is lost.

* **void decreaseProtestorCount();**

Reduces the count of Protestors in the game.

**Missing Functionality & Bugs**

* When I followed the spec’s requirement that a Protestor’s *ticksToWaitBetweenMoves= max(3-level/4,0)*, the Protestors all moved too fast compared to the TunnelMan, which made it difficult to escape from them. My TunnelMan seemed to be moving at the same speed as the one in the sample game, but the Protestors were way faster than those in the sample. I was unfortunately unable to resolve this issue even after hours of trouble-shooting. So, as shown in the below screenshot of my *Protestor* constructor, I set *ticksToWaitBetweenMoves* to 5 for the sake of more normal gameplay.
* 
* I was unable to do part 5 of the Hardcore Protestor’s implementation, so I am treating the Hardcore Protestors like Regular Protestors.
* For some reason that I am not certain of, my *Protestor* queue-based maze-searching algorithm (please see *Protestor::pathSearcher()* ) works only when the Protestor is squirted into submission at the very top of the game board (above any Earth objects). If the Protestor is anywhere else, it ends up remaining permanently frozen on the screen rather than leaving.
* The Protestors sometimes freeze in their tracks, and I was unable to resolve this issue.

**Design Decisions/Assumptions**

I tried following the part of the spec which states that “the TunnelMancannot occupy a square that is less than or equal to a radius of 3 away from the center of any Boulder*”*. However, this caused my TunnelMan to be unable to get as close to the Boulder as it can in the sample. Consequently, I created four helper functions that check whether there is part of a boulder to the east, south, west or north of the TunnelMan. I was then able to make the TunnelMan move very similarly to the one in the sample game.

**How I Tested Each Class**

The Actor class, being an abstract base class, could not be tested directly. I knew that it worked because all game actors were successfully constructed. Moreover, the member functions *setAlive,* *getAlive, getWorld, setActorVisible* and *getActorVisible* were used extensively in derived classes and in the *StudentWorld* class.

The Earth class was tested by creating a 64 by 60 2D array of Earth pointers in StudentWorld::init(). Earth objects had to be successfully constructed in order for the game board (especially with the vertical shaft down the middle) to be properly displayed. The Earth destructor was tested via the StudentWorld class’s *cleanUp* function, which had to delete all the dynamically allocated Earth objects from StudentWorld::init().

The Person class was proven via the successful construction and destruction of the TunnelMan, Regular Protestors and Hardcore Protestors. The class’s isAnnoyable() public identifier function was successfully used by StudentWorld’s function to search for Protestor’s in a particular area of the game board. The getHitPoints and setHitPoints methods were also successfully used to damage the TunnelMan and protestors.

The TunnelMan class was tested via the successful construction of the TunnelMan object and its successful destruction upon deletion in StudentWorld::cleanUp. TunnelMan’s accessor and mutator functions correctly returned and altered the amounts of gold nuggets, squirts and sonar kits in its possession. The class’s doSomething() method successfully called upon the StudentWorld object to make Earth objects in the TunnelMan’s vicinity invisible, checked to see if the human player pressed a key, made the TunnelMan fire a squirt (if applicable), and moved the TunnelMan. The TunnelMan could move around without leaving the Earth board’s perimeter or going through boulders.

The Protestor class was tested via the successful construction and appearance on screen of Hardcore and Regular Protestors. The StudentWorld class could use its annoyProtestor public member function to damage a Protestor depending on whether it was bonked by a boulder or squirted on. The bribeProtestor method correctly incremented the player’s score and forced a Hardcore Protestor to pause for a few moments once it picked up a golden nugget dropped by the TunnelMan. This was verified through repeated testing of the game and checking the game status line at the top. The protected member functions were checked by seeing whether the Protestors were able to navigate around the screen on their own, at random and avoiding Earth.

The Regular Protestor and Hardcore Protestor classes were tested in the exact same way because I was unable to implement the latter’s triangulation algorithm. I checked that they were able to shout at the TunnelMan and decrease his health. I also confirmed that they were able to be bonked by Boulders.

The Goodies class was proven via the successful construction and destruction of Barrels, SonarKits, Gold Nuggets and Water. A Goodie could be set to be temporary or permanent (depending on the context) and pickup-able or not pickup-able by the TunnelMan. A Goodie’s tick lifetime could also be successfully set.

The Barrel class was tested by checking that if a Barrel is not currently visible and the TunnelMan is within a radius of 4.0 from it, then the Barrel makes itself visible. The Barrel::doSomething() method successfully informed the StudentWorld object that it was picked up, which caused the game status line to be updated (“oil left” reduced by one).

The SonarKit class was tested by checking that the SonarKit could successfully appear at the top-left corner of the game screen and disappear after a while. I verified that the SonarKit::doSomething() function worked by making sure that once the TunnelMan got within a radius of 3 from a Sonar Kit, the TunnelMan’s inventory of sonar kits would increase by 1 (as shown in the game status line). I also knew that the SonarKit class worked because the Sonar Kit would appear and disappear spontaneously.

The GoldNugget class was tested via the successful construction and destruction of Gold Nugget objects in various contexts. Gold Nuggets that were randomly distributed at the very beginning of the game were initially invisible. I could set Gold Nuggets to be temporary or permanent and pickup-able or not pickup-able by the TunnelMan. I checked that the game status line would be updated when a Gold Nugget was close enough to the TunnelMan.

I tested the Water class by checking that they would occur in passageways (no Earth) spontaneously throughout gameplay. I also verified that the Squirt inventory and game status line at the top were updated whenever the TunnelMan approached a Water image.

I tested the Boulder class by checking that they could drop and disappear or bonk the TunnelMan or a protestor. Boulders could only fall if all four squares of Earth underneath them were dug away. I also verified that Boulders were in a four by four square without any Earth and that the correct number of Boulders were allocated to each level.

I tested the Squirt class by verifying that the TunnelMan could successfully shoot water and that a Squirt could die after moving forward four squares (in the same direction that the TunnelMan was facing) or earlier if it hits Earth or a Protestor. I also checked that Protestors could be sufficiently annoyed to stop harassing the TunnelMan.

Last but not least, the StudentWorld class was proven successful through repeated playing of the game. The init, move and cleanUp methods had to work for the game to be setup, run, and terminate without a memory leak. StudentWorld’s member functions successfully manipulated its member objects and assisted Actor derived class’s operations.