==================================

Name: Nevin Liang

UCLA ID: 705-575-353

Subject: Project 4 Writeup

Date: August 21, 2020

==================================

Design Description for Public Member Funcs:

-------------------------------------------

Actor(**int** imageID, **int** startX, **int** startY, Direction dir = right,

**double** size = 1.0, **unsigned** **int** depth = 0);

* This function was the constructor so I had to implement it in the public interface of the class

**virtual** ~Actor();

* I chose to implement this destructor virtually because it is standard practice (as I learned in the lectures), to set all destructors to be virtual and public.

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

* I chose to implement this as a pure virtual function because actor does not do anything and no instance of the actor object needs to be created.

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

* I chose to make this one public because a lot of functions outside of this class had to be able to use it. I could have made it protected, but public was the easiest and safest to do so. Children needed to use it a lot on each other as well

**void** setWorld(StudentWorld\* ptr);

* I chose to make this one public because it had to be called from a different class.

**virtual** std::string getType() = 0;

* I chose to make this one public and a pure virtual function because it doesn’t matter what the type is for Actor because literally every object is of type actor class. All of the children functions would fill it in later.

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

* I chose to make this one virtual and public. (Virtual) because its children classes/sub classes would be able to change it and use it to do different things. I also made isLive public because I needed it in other classes.

Earth(**int** startX, **int** startY);

* The function was the constructor so I had to implement it in the public interface of the class

**virtual** ~Earth();

* I chose to implement this destructor virtually because it is standard practice (as I learned in the lectures), to set all destructors to be virtual and public.

**virtual** **void** doSomething();

* I chose to implement this as a virtual function because Earth has its own specific implementation of dosomething that is different from other child classes of actor’s virtual function.

**virtual** std::string getType();

* I chose to make this public because a lot of other classes need to access it, especially its siblings.

**virtual** **bool** isLive();

* I chose to make this public because a lot of other classes need to access the live function and I made it virtual because it inherited and changed the details from its parent class.

TunnelMan();

* The function was the constructor so I had to implement it in the public interface of the class

**virtual** ~TunnelMan();

* I chose to implement this destructor virtually because it is standard practice (as I learned in the lectures), to set all destructors to be virtual and public.

**virtual** **void** doSomething();

* I chose to implement this as a virtual function because Tunnelman has its own specific implementation of dosomething that is different from other child classes of actor’s virtual function.

**void** annoy(**int** amount);

* I chose to implement this publicly because a lot of other classes have to be able to annoy the tunnelman, especially stuff like protestors and boulders which are not up or down related.

**int** getHP();

* I made the accessors public because it was much easier to use them in other classes and files like studentworld

**virtual** std::string getType();

**virtual** **bool** isLive();

**void** pickOilUp();

* I made this one public because I needed to be able to access it from the studentworld class which was notified when the barrel was picked up by the barrel.

**int** getOil();

* I made the accessors public because it was much easier to use them in other classes and files like studentworld

**int** getWater();

**int** getNuggets();

**int** getSonar();

Boulder(**int** startX, **int** startY);

* The function was the constructor so I had to implement it in the public interface of the class

**virtual** ~Boulder();

* I chose to implement this destructor virtually because it is standard practice (as I learned in the lectures), to set all destructors to be virtual and public.

**virtual** **void** doSomething();

* I chose to implement this as a virtual function because Boulder has its own specific implementation of dosomething that is different from other child classes of actor’s virtual function.

**void** drop();

* I made this one not virtual because it does not inherit from a parent class and does not override anything

**virtual** std::string getType();

* Accessor so made it public and virtual because inherits from parent class

**virtual** **bool** isLive();

* Same because it is accessor and inheriting from parent class

Goodies(**int** imageID, **int** startX, **int** startY, Direction dir,

**double** size, **unsigned** **int** depth);

* The function was the constructor so I had to implement it in the public interface of the class

**virtual** ~Goodies();

* I chose to implement this destructor virtually because it is standard practice (as I learned in the lectures), to set all destructors to be virtual and public.

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

* I chose to make this one public and a pure virtual function because it doesn’t matter what the type is for Goodies because literally every object is of type Goodies class. All of the children functions would fill it in later.

**virtual** std::string getType();

* Accessor so it is public and other classes have to get it and virtual because inherits from parent function

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

* I chose to make this one public and a pure virtual function because it doesn’t matter what the type is for Goodies because literally every object is of type Goodies class. All of the children functions would fill it in later.

Barrels(**int** startX, **int** startY);

* The function was the constructor so I had to implement it in the public interface of the class

**virtual** ~Barrels();

* I chose to implement this destructor virtually because it is standard practice (as I learned in the lectures), to set all destructors to be virtual and public.

**virtual** **void** doSomething();

* I chose to make this one public and a pure virtual function because it doesn’t matter what the type is for Goodies because literally every object is of type Goodies class. All of the children functions would fill it in later.

**virtual** std::string getType();

* Accessor so it is public and other classes have to get it and virtual because inherits from parent function

**virtual** **bool** isLive();

* I chose to make this one public and a virtual function because it inherits from the parent class.

StudentWorld(std::string assetDir);

* The function was the constructor so I had to implement it in the public interface of the class

**virtual** **int** init();

* The function was the initializing one so I made it public because it was very important and that it was called first by the gameobject class.

**virtual** ~StudentWorld();

* I chose to implement this destructor virtually because it is standard practice (as I learned in the lectures), to set all destructors to be virtual and public.

**virtual** **int** move();

* This function was implemented virtually because it was a redefined function from a parent class

**virtual** **void** cleanUp();

* This function was defined virtually because it was also a redefined function from a parent class

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

* I defined areaNoEarth publicly because a lot of the classes from the Actor set of classes had to call this function

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

* I defined areaNoEarth publicly because a lot of the classes from the Actor set of classes had to call this function

**void** makeBoulders();

* I defined this one publicly because it made it easier to test and see from the Actor class exactly what was going on with the files.

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

* I implemented pointNoEarth publicly because a lot of the classes from the Actor set of classes had to call this function

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

* A function from Actor had to use this class in Student World so I made it public.

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

* I implemented checkBoulderHitMan publicly because one of the classes from the Actor set of classes had to call this function

**void** addBarrelToPlayer();

* I implemented pointNoEarth publicly because a lot of the classes from the Actor set of classes had to call this function

**double** tunnelManDistance(**int** x, **int** y);

* This one was a public function because a ton of classes in Actor had to use this function. Especially the boulder class.

**void** makeBarrels();

* I defined this one publicly because it made it easier to test and see from the Actor class exactly what was going on with the files.

**void** statusbarfunc();

* I made this one public because I could test it from other areas of my code and make sure that it worked. It was not virtual because it did not inherit from any parent class.

Incompleted Functionality:

-------------------------------------------

I wasn’t able to complete the squirt water classes and algorithms for the TunnelMan, or the Hardcord Protestor. I attempted to code the Protestor and it works. I did not code all of the goodies, and was not able to finish gold nuggets, water, or sonar. However, I did do Barrels and the level up functions which were the backbone of the game.

Design Assumptions I Made:

--------------------------------------------

I made the assumption that 6 blocks away between objects like boulders and barrels was 6 blocks between the x and y coordinates for each, so they looked pretty close together sometimes because they are technically 2 blocks apart. This could have been interpreted as 6 spaces in between in which I would have to change my 6 to a 10.

I also assumed that the barrel and boulder could not spawn at the very top or the shaft which makes sense because then initial game states would be slightly strange.

Description of how I tested each of my classes:

---------------------------------------------

Actor:

* This class was not tested with hard-code because it is an abstract class so I can not make an object with it. However, the other classes that inherit from it working are good enough.

Earth:

* This class was tested by making a large array of them in the StudentWorld class. I had to manipulate every function that was in Earth from the accessors (which had a high chance of working) to the functions that created and deleted them (which had a much lower chance of working :))
* I ended up having to change multiple instances of functions in my class because I read on in the specs and I was able to reuse many of the functions elsewhere.

TunnelMan:

* This one was by far the steepest learning curve when writing the class, so I tested this one as I went, which slowed me down a bunch. However, in order to test this, in part 1 I was able to test this a lot because all I had in the entire screen was just a single tunnelman. I tested this repeatedly after I finished boulders by making the tunnelman go towards it from all directions, seeing if it would hit the boulders or go through them. I also tested whether my man could pick up barrels and get crushed by boulders.
* I also tried escaping the boundaries of the earth map and since I did boundary checking this did not work, so my program code was correct!

Boulder:

* I mainly tested this one out together with TunnelMan it was basically testing what cases Boulder would fall. Initially I read the specs wrong and the boulder fell when only a single earth block was absent below it, but testing it out made me realize what I was doing wrong and I quickly changed it.

Goodies:

* This class was not tested with hard-code because it is an abstract class so I can not make an object with it. However, the other classes that inherit from it working are good enough.

Barrels:

* This was the barrel class which was probably the hardest object to code aside from tunnelman. It had to do a lot of stuff such as making a sound, incrementing tunnel player’s points, and disappearing and appearing throughout the game. I tested this like I tested all the other classes, in that I just walked around the barrels trying to find them and when I got within 4 blocks it appeared. Getting within 3 blocks, I collected it.

StudentWorld:

* This was the biggest class, and the most complex, but not the hardest to code algorithmically. It incorporated a lot of helper functions that I had to test on the side with random inputs I created because they were very disjoint. For example, a lot of the functions did totally separate things with totally separate classes. I ran Tunnel man’s game multiple times to test my code for that class, and tried breaking the StudentWorld class, but nothing works :).