**Project 3 Report**

1. Description of public member functions:

Actor.h Functions:

Actor(int imageID, int startX, int startY, StudentWorld\* ptr, Direction dir = none, int hp = 0);

Constructor for actor, sets data members.

Constructor doesn’t need to be virtual.

virtual ~Actor(){};

Destructor for actor.

Base class destructor should be virtual in case derived classes must override

virtual void doSomething()=0;

Function tells each actor what to do during each tick.

Pure virtual because individual actors take different actions.

Define in base class so that StudentWorld knows all derived actors can doSomething (to make function callable by iterator)

virtual bool attackable() = 0;

Tells caller (bullet) whether actor can be hit

Pure virtual because some actors can be hit and some can’t.

Define in base class so that it is accessible by a general Actor iterator/pointer.

virtual bool blockRobotAttack() = 0;

Function returns whether actor can block a robot attack.

Pure virtual because this varies. For ex: boulders can be attacked by players but not by shooting robots.

Define in base class so that it is accessible by a general Actor iterator/pointer.

virtual bool blockCoord(Direction dir, Actor\* callerID){ return true; };

Function returns whether actor blocks other actors from moving onto its spot. It takes a pointer to its caller, because some objects only block specific actors and not others who might also call this function.

Function is virtual because a lot of actors do block other actors, but a few do not, so they can override and return true.

Define in base class because all actors need this function. They either block or don’t.

virtual bool countsInFactoryCensus(){ return false; };

Function returns whether actor is any type of kleptobot, which would count in a factory census.

Virtual because a function in StudentWorld will call this when iterating through all actors.

Define in base class because all actors must have this function. Most will return false, but kleptobots must be able to override.

StudentWorld\* getWorld(){ return m\_world; };

Return a pointer to StudentWorld.

Non-virtual because no actor should need to override. Define in base class because all actors need to reference StudentWorld.

void nextCoord(int& x, int& y, Direction& dir);

Calculate the next coordinate to move to based on x, y and direction.

Non-virtual because it works the same for all actors.

Define in base class because most actors need to calculate the next coordinate at some point.

void fire();

Fire a bullet.

Non-virtual because bullets are fired in one way.

Define in base class because multiple actors need to fire bullets. The ones that don’t will never call this function.

bool alive(){ return m\_alive; };

Return whether actor is alive.

Non-virtual because all actors are either alive or dead.

Define in base class because most actors can die in some way and should know if they are dead.

void setDead(){ m\_alive = false; };

Make an actor dead.

Non-virtual because all actors die in same way.

Define in base class because most actors can be killed.

void decHP(int n){ m\_hp -= n; };

Damage an actor by a certain amount.

Non-virtual because all actor are damaged by reducing hp, if damageable.

Define in base class because multiple actors can be damaged.

int checkHP(){ return m\_hp; };

Check health.

Non-virtual because same for all actors.

Define in base class because multiple actors might need it.

void restoreHP(){ m\_hp = 20; };

Restore player’s health.

Non-virtual because no other way to restore health.

Define in base class because base needs to access private m\_hp

int checkAmmo(){ return m\_ammo; };

Check ammo left.

Non-virtual because no need to override.

Define in base because base needs to access m\_ammo.

//reduce ammo by one when bullet fired

void decAmmo(){ m\_ammo--; };

//add ammo when pick up ammo goodie

void addAmmo(){ m\_ammo += 20; };

Player Functions:

Player(int startX, int startY, StudentWorld\* ptr);

Player constructor. No need to be virtual.

virtual void doSomething();

Each tick, check if player has pressed a key, then move or shoot accordingly.

Virtual because player does not act the same as other actors during a tick.

Must define in derived Player class to specify player’s actions.

virtual bool attackable();

Return true because player is attackable by bullets. Decrease HP and set dead if needed.

Define in player class because this function is pure virtual in Actor class.

virtual bool blockRobotAttack(){ return false; };

Return the fact that players can’t block robot attacks.

Define here because function is pure virtual in base Actor class.

void addAmmo(){ m\_ammo += 20; };

Add ammo to player after picking up an ammo goodie.

Non-virtual because only player needs this function.

int checkAmmo(){ return m\_ammo; };

Check ammo left.

Non-virtual because only player has limited ammo.

Wall Functions:

Wall(int startX, int startY, StudentWorld\* ptr);

Constructor does not need to be virtual.

virtual void doSomething(){};

Walls should do nothing, so function is empty.

Function must still be implemented because it is pure virtual in base Actor class and will be called by StudentWorld for every actor.

virtual bool attackable(){ return true; };

Walls can be hit by bullets. But no damage, just return true.

Function must be implemented because pure virtual in base Actor class.

virtual bool blockRobotAttack(){ return true; };

Walls block robot attacks.

Function must be implemented here because it is pure virtual in base Actor class.

Bullet Functions:

Bullet(int startX, int startY, StudentWorld\* ptr, Direction dir);

Bullet constructor, no need to be virtual.

virtual void doSomething();

Bullet should check if it can move or if there is anything that can be hit on its coordinate.

Function must be implemented here because it is pure virtual in base Actor class.

virtual bool attackable(){ return false; };

Bullets are not attackable.

Implement here because function is pure virtual in base Actor class.

virtual bool blockCoord(Direction dir, Actor\* callerID){ return false; };

Bullets don’t block actors from running into them.

Implement virtual function here to override original function in base Actor class.

virtual bool blockRobotAttack(){ return false; };

Bullets don’t block Robots from attacking.

Implement here because function is pure virtual in base Actor class.

Boulder Functions:

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

Boulder constructor should not be virtual.

virtual void doSomething(){};

Boulders do nothing.

Virtual function to override pure virtual function in base Actor class.

virtual bool attackable();

Boulders are attackable by player. Decrease hp and check whether it is dead.

Virtual function must be implemented here because function is pure virtual in base Actor class.

virtual bool blockCoord(Direction dir, Actor\*);

Boulders may be pushed by actors but not by anything else. Boulders can also be on the same coordinate as a hole. Check these cases.

Must override virtual function in base class because boulders do not always block.

virtual bool blockRobotAttack(){ return true; };

Boulders do block attacks from robots. Only the player can shoot boulders.

Function is implemented here because it is pure virtual in base Actor class.

bool push(Direction dir);

Check whether boulder can be pushed in a direction.

Non-virtual because only boulders can be pushed.

Hole Functions:

Hole(int startX, int startY, StudentWorld\* ptr);

Hole constructor.

virtual void doSomething();

Holes should check whether there is a boulder on them. If so, set dead.

Function must be implemented in hole class because it is pure virtual in base Actor class.

virtual bool attackable(){ return false; };

Holes can’t be attacked by bullets.

Function must be implemented in hole class because it is pure virtual in base Actor class.

virtual bool blockCoord(Direction dir, Actor\* callerID);

Holes do not block boulders. Check this case.

Function here must override the virtual function in base Actor class.

virtual bool blockRobotAttack(){ return false; };

Holes do not block robots from shooting.

Function must be implemented in hole class because it is pure virtual in base Actor class.

Exit Functions:

Exit(int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual void doSomething();

Exit should check if player is on its coord. If so, level complete. Otherwise, check whether it should reveal itself.

Function must be implemented in exit class because it is pure virtual in base Actor class.

virtual bool attackable(){ return false; };

Exit can’t be attacked by bullets.

Function must be implemented in exit class because it is pure virtual in base Actor class.

virtual bool blockCoord(Direction dir, Actor\* callerID){ return false; };

Exits do not block anything.

Implement here to override virtual function in base class.

virtual bool blockRobotAttack(){ return false; };

Exits do not block robots from attacking.

Function must be implemented in exit class because it is pure virtual in base Actor class.

Item Functions:

Item(int imageID, int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual void doSomething();

Items should check whether player is on same square. If so, item must be picked up and set dead.

Function must be implemented in exit class because it is pure virtual in base Actor class.

virtual bool attackable(){ return false; }

Items can’t be attacked by bullets.

Function must be implemented here because it is pure virtual in base Actor class.

virtual bool blockCoord(Direction dir, Actor\* callerID);

Items don’t block player or robots. Check these cases.

Function must override virtual function in base Actor class.

virtual void collectItem() = 0;

Virtual because all items have different effects when collected, but all are collectible so function should exist as pure virtual in base Item class.

virtual bool blockRobotAttack(){ return false; };

Items all can’t block robot attacks.

Implement function in Item class, so it has the same effect for all items.

Jewel Functions:

Jewel(int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual void collectItem();

Jewels give 50 points, and StudentWorld must know there are fewer jewels left.

Implement here because function is pure virtual in base Item class.

Goodie Functions:

Goodie(int imageID, int startX, int startY, StudentWorld\* ptr);

Constructor.

void dropItem(int x, int y);

Make item visible and move to robot’s coordinate when robot dies and drops it.

Non-virtual because all goodies can be dropped in the same way.

Extra Life Functions:

Extra\_Life(int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual void collectItem();

Increase score and lives.

Implement base Item class’s pure virtual function.

Restore Health Functions:

Restore\_Health(int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual void collectItem();

Restore player’s health and increase score.

Implement base Item class’s pure virtual function.

Ammo Functions:

Ammo(int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual void collectItem();

Add to player’s ammo and increase score.

Implement base Item class’s pure virtual function.

Robot Functions:

Robot(int imageID, int startX, int startY, StudentWorld\* ptr, Direction dir, int hp);

Constructor.

virtual void doSomething();

Robots must try to move/shoot.

Virtual function because some robots may behave differently.

virtual bool attackable();

Robots are attackable. Decrease hp and check whether robot is dead.

Function must be implemented here because it is pure virtual in base Actor class.

virtual bool blockRobotAttack(){ return true; };

Robots block attacks by other robots.

Function must be implemented here because it is pure virtual in base Actor class.

virtual bool decideShoot();

Decide whether there is a clear path to the player to shoot.

Include in Robot class because multiple robot types can shoot.

Virtual because some robots can’t shoot at all, so function must be override-able.

void reverseDirection();

Reverse robot’s direction.

Non-virtual because this should be the same for all robots.

bool act();

Robot should decide whether to act based on ticks.

bool attemptMove(Direction dir);

Check whether robot can move to coordinate.

Non-virtual because all robots use this function the same way.

GraphObject::Direction translateDirection(int d);

Translate direction from int to Direction type.

Non-virtual because all robots use this function the same way.

Include in robot class so all robots can use it.

void tickReset(){ m\_tick = 1; };

Reset robot’s ticks to 1.

Non-virtual because this is the same for all robots.

virtual int pointsWorth() = 0;

Return points that a robot is worth if killed.

Pure virtual because robots are worth different points.

Horiz\_Snarlbot Functions:

Horiz\_Snarlbot(int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual int pointsWorth(){ return 100; };

Snarlbot worth 100 points.

Implement here because function in Robot class is pure virtual.

Vert\_Snarlbot Functions:

Vert\_Snarlbot(int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual int pointsWorth(){ return 100; };

Snarlbot is worth 100 points.

Implement here because function in Robot class is pure virtual.

Kleptobot Functions:

Kleptobot(int startX, int startY, StudentWorld\* ptr, int imageID = IID\_KLEPTOBOT, int hp = 5);

Constructor.

virtual ~Kleptobot();

Destructor to drop goodie if kleptobot is killed and had a goodie.

Virtual to override base Actor destructor and specify special case for kleptobots.

virtual bool decideShoot(){ return false; };

Regular kleptobots don’t shoot.

Virtual because angry kleptobots will need to override.

virtual void doSomething();

Kleptobots should decide whether to shoot or pick up goodies.

Implement here because function is pure virtual in base Actor class.

virtual int pointsWorth(){ return 10; };

Kleptobots are worth 10 points.

Implement here because function is pure virtual in base Robot class.

virtual bool countsInFactoryCensus(){ return true; };

Return true because kleptobots count toward factory census.

Virtual function to override function in base Actor class.

Angry Kleptobot Functions:

Angry\_Kleptobot(int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual bool decideShoot(){return Robot::decideShoot(); };

Angry kleptobots must decide whether to shoot. Use the Robot class’s decideShoot function, because regular kleptobots can’t shoot.

Virtual to override regular kleptobot’s decideShoot function.

virtual int pointsWorth(){ return 20; };

Angry Kleptobots are worth 20 points.

Implement here because function is pure virtual in base Robot class.

Kleptobot Factor Functions:

Kleptobot\_Factory(ProductType type, int startX, int startY, StudentWorld\* ptr);

Constructor.

virtual void doSomething();

Factories should decide whether to produce kleptobots depending on number of kleptobots nearby.

Implement here because function is pure virtual in base Actor class.

virtual bool attackable(){ return true; };

Factories can be hit by bullets but are not damageable.

Implement here because function is pure virtual in base Actor class.

virtual bool blockRobotAttack(){ return true; };

Factories block robots from attacking.

Implement here because function is pure virtual in base Actor class.

1. List of unfinished functionality:

None.

1. Design decisions and assumptions:

For some functions that the sample interface implemented in the StudentWorld file, I decided to implement them in the Actor file. I think it still works. For example, increaseAmmo() and restorePlayerHealth().

I also wasn’t entirely sure when/where to make goodies reappear after kleptobots died. I decided to place the code for making kleptobots drop goodies in a separate destructor, so goodies would reappear at the kleptobot’s coordinate after the kleptobot carrying it was destroyed.

1. Testing:

Player class:

I tested the player class by moving around and shooting objects that could and could not be shot. I made sure the player could pick up the correct items and get points for them. The player finished levels correctly and restarted levels correctly. I checked that the number of lives left worked correctly. The player could be killed or shot at correctly by robots.

Wall class:

I made sure the walls showed up at the correct locations and blocked bullets and other actors. The walls did nothing else.

Bullet class:

I checked that the bullets damaged the player and robots correctly. Bullets passed through other items. Bullets were blocked by walls and factories. Actors could run into bullets and die. After ammo == 0, no more bullets can be shot.

Boulder class:

I checked that boulders could be pushed and destroyed. A boulder could be on the same spot as a hole, after which both hole and boulder would disappear. I checked that only the player could push the boulders and they were never pushed by robots or other things.

Hole class:

I checked that holes could not be walked across by the player or robots. I tried pushing boulders into them to make sure they disappeared. I shot bullets across them to make sure they did not block.

Exit class:

I made sure the exit appeared at the right time after all jewels were collected. I checked that the player and robots could walk across the exit. When the player touches the exit, the level finishes. I checked that the exit appearing sound played only once and did not repeat.

Jewel class:

I checked that the player could pick up jewels and the exit appeared when all jewels were gone. I checked that no robots could pick up jewels. I made sure bullets could cross jewels and actors could walk across them. I checked that collecting jewels increased the score by the right amount.

Goodie class:

I checked that players could pick up goodies as well as kleptobots. I checked that goodies were dropped properly on the correct location and made sure the player couldn’t pick them up while a kleptobot had them. I fixed a problem with the kleptobots so that the kleptobots also couldn’t pick up invisible items that another kleptobot already had. I created a level with just goodies to test that they gave the correct points and had the right effects on health/ammo etc.

Horiz\_Snarlbot and Vert\_Snarlbot classes:

I checked that they shot at the player when I walked in front of them. I made sure they didn’t shoot at the player if they were turned away or if there was a blocking object in between the player and the snarlbot. I tried shooting them and made sure they could die correctly. I stepped through various functions to check that they moved at the right speed on the correct ticks.

Kleptobot class:

I estimated that they seemed to be moving correctly by comparing against the sample game. I created levels where objects were placed in front of the kleptobots to check that they collected goodies correctly. When the player killed a kleptobot that had collected a goodie, the goodie was dropped at the correct location when the kleptobot died. I checked that regular kleptobots wouldn’t shoot when the player walked in front of them.

Angry Kleptobot class:

I tested whether angry kleptobots would shoot if the player stepped in front of them. I added a factory in front of the player, then stepped through the code to make sure angry kleptobots were calling the right functions, because I originally made a mistake by forgetting to override the regular kleptobot’s decideShoot function. I made sure they could still pick up objects and could still be killed. I estimated how they moved compared to the sample game, and they seemed to move reasonably. I checked that angry kleptobots behaved like regular kleptobots except for the shooting ability. They dropped goodies correctly, etc.

Kleptobot Factory class:

I stepped through the code to make sure factories were creating the correct type of kleptobots at a reasonable rate. I checked that factories could block bullets, and robots couldn’t shoot through them. I checked that the player and other robots weren’t walking over the factories and that only newly created kleptobots ever appeared on the same coordinate.

StudentWorld class:

I stepped through the init, cleanUp, and move functions. I checked that the text displayed and updated in a format that matched the sample game. I tried losing lives to make sure the level cleaned up and restarted correctly. I tried advancing levels to make sure it worked.