Daniel Frees

02/26/20

**CS32 Proj 3 Report**

**High-Level Description of Public Member Functions in Each Class:**

**StudentWorld: public GameWorld)**

*Constructor:* StudentWorld has a constructor which takes an assetPath parameter (used for connecting correctly to all the graphics and sounds etc. on the computer). It initializes the m\_racer member (a pointer to GhostRacer) to nullptr. It also initializes m\_souls2Save, m\_bonusPoints, and m\_highestWhiteBorder to 0. The constructor is in StudentWorld because every class must have a constructor, and it is not virtual because constructors cannot be virtual. **This reasoning holds true for all my constructors throughout this report.**

*Destructor:* StudentWorld’s destructor simply calls cleanUp(), which is detailed below. It is virtual because this is good practice for a derived class, even though we don’t have any more deeply derived world classes in this project.

*Virtual int init():*

This function initializes the data structures keeping track of the level (ie: souls2Save and bonus points). It also inserts a new ghost racer and gets border lines set up. It returns a status for the game to continue when it is done. This function is in the studentworld class because it initializes the StudentWorld, and it is virtual because it redefines a virtual method from the GameWorld class.

*Virtual int move():*

This function runs a tick of the game by giving each actor a chance to do something, checking whether the level is complete (all necessary souls saved)/ game is over (racer is dead), removing dead actors, potentially adding new actors, updating the gamestatus line, decrementing bonus points, and returns a value indicating whether the level is complete, player died, and level was finished. It is defined in StudentWorld because StudentWorld handles all of these things which happen each tick, and it is virtual because it is a re-implementation of a virtual method from the GameWorld class.

*Virtual void cleanUp():*

This function removes all actors that are still alive after completing a level or dying, so that things can start anew on a new level or things can be cleaned up after the game is over. It is also called via the destructor so that all memory is freed in case of an early quit from the game.

*Double getRacerSpeed() const:*

This is a simple getter which returns the racer’s vertical speed. Racer getters such as this are implemented in StudentWorld because many kinds of Actors need to ask the world about the racer and I preferred to make public getters rather than a function with returns a pointer to the racer. This function is not virtual because it is not re-implemented anywhere.

*Int getRacerDirection() const:*

This is a simple getter which returns the racer’s direction. Racer getters such as this are implemented in StudentWorld because many kinds of Actors need to ask the world about the racer and I preferred to make public getters rather than a function with returns a pointer to the racer. This function is not virtual because it is not re-implemented anywhere.

*Double getRacerX() const:*

This is a simple getter which returns the racer’s x coordinate. Same class placement/ non-virtual reasoning as above.

*Double getRacerY() const:*

This is a simple getter which returns the racer’s Y coordinate. Same class placement/ non-virtual reasoning as above.

*Bool racerIsAlive() const*:

This is a simple getter for the racer’s life status. Same class placement/ non-virtual reasoning as above.

*Void killRacer():*

This is a simple setter to kill the racer. Some actors need to kill the racer and I preferred to make public setters rather than a function returning a pointer to the racer. This function is not virtual because it isn’t re-implemented nor does it need to be for any derived classes.

*Void damageRacer(const int& amt):*

This is a simple setter to damage the racer by a certain amount. Same reasoning as above for for why this is hosted by the StudentWorld class and isn’t virtual.

*Void spinRacer():*

This is a simple setter which calls the racer’s getSpun() method so that Oil Slicks can spin the racer without asking for a pointer to her. StudentWorld handles all racer setting interactions such as this. It is not virtual because this method never needs to be re-implemented.

*Void healRacer(const int& amt):*

This is a simple setter which increases the racer’s health by a certain amount. It allows healing goodies to heal the racer via the Student World. Same reasoning as above for why this is hosted by the StudentWorld class and isn’t virtual.

*Void giveRacerAmmo(const int& amt):*

This is a simple setter method which increases the racer’s ammo by a certain amount. Same reasoning as above for why this is hosted by the StudentWorld class and isn’t virtual.

*Void saveASoul():*

This is a simple setter method which decrements the souls left to be saved so that the StudentWorld can keep track of progress throughout the level. This function is hosted in StudentWorld because StudentWorld handles tracking level progress, and it is not virtual because this method never needs to be re-implemented in a derived class.

*Void addActor(Actor\* a):*

This is a method which allows an actor pointer to be passed to the world and added into the m\_actors actor container (this method is used by actors when they want to instantiate a new actor where they died, for example). This is in StudentWorld because StudentWorld should handle its own container of actors, and it is not virtual because the method never needs to be re-implemented in this project setup.

*Bool overlapWithRacer(const Actor\* a) const:*

This method calls a private helper method and checks whether the passed actor overlaps with the racer. The private theyOverlap() helper method simply follows the pseudocode provided in the spec to check whether two actor objects overlap. It returns true if the passed actor overlaps with the racer. It is in the StudentWorld because it makes sense for the StudentWorld to handle checks such as these between its various actors (including the GhostRacer). It is not virtual because it need not be re-implemented.

*Void closestCAWActorsInLane(const Actor\* a, double& pixelsFront, double& pixelsBack) const:*

Here is pseudocode for this function:

Get the passed actor’s lane (uses a private helper function where I determine an actor’s lane using its current X coord)

Set pixelsFront to VIEW\_HEIGHT and pixelsBack to -VIEW\_HEIGHT as default values in case we find no CAW actors in front or behind in the same lane as Actor a.

For every actor in the actor container

If this actor (actor b) is collision avoidance worthy and alive and is not the same as the actor a passed in (aliasing check)

If actor b is in the same lane as actor a

pixelsBetween = actor b’s y value – actor a’s y value

if pixelsBetwen is a positive number and is smaller than pixelsFront, set pixelsFront to this new value

if pixelsBetween is a negative number and is greater than pixelsBack, set pixelsBack to this new value

After finishing running, the passed doubles for pixelsFront and pixelsBack hold the distances until the nearest CAW actor to Actor a in front and behind it in its lane (or VIEW\_HEIGHT and -VIEW\_HEIGHT if no relevant CAW actors). This is a useful method for Zombie Cabs so that they can implement their movement planning and slow down or speed up accordingly for non-Ghost Racer CAW Actors in their lane.

This method belongs in StudentWorld because it requires an overarching understanding of where all the actors are and StudentWorld handles such things. It is not virtual because there is no need to re-implement it.

*Bool projectileMaybeDamageActor(const Actor\* projectile) const:*

Pseudocode for this method:

For all actors in m\_actors

If the actor is not the passed projectile and the actor is affected by projectiles and the actor is alive and the projectile overlaps this actor

Hit the actor with an amount of 1 (1 is the potential damage the actor might take)

Return true

Return false

*Notable Protected/ Private Methods\*:*

\*requested by Prof. Smallberg. Much of my functionality is in protected/private methods so he said to make note of the significant functions that are protected/private.

*bool theyOverlap(const Actor\* a, const Actor\* b) const:*

This private helper method finds whether two objects overlap as outlined in the spec. This is handled by the world because the world knows where both actors are. It is not virtual because doesn’t need to be re-implemented.

*Int getActorLane(const Actor\* a) const:*

I implemented this private helper to help with concisely determining actor lines. Used in closestCAWActorsInLane for example.

Here is the pseudo:

Initialize the return value laneNumber to -1

If the passed actor has an X value >= the left edge of the road and less than the left white divider

laneNumber = 1

else If the passed actor has an X value >= left white divider and < the right white divider

laneNumber = 2

else If the passed actor has an X value >=the right white divier and less than the right edge of the road

laneNumber = 3

return laneNumber

This function returns 1 if the passed actor is in the left lane, 2 if in the middle lane, and 3 if in the right lane. It is defined in StudentWorld because the world can easily handle comparing locations of actors with the lane setup of the world. It is not virtual because there is no need to re-implement this function.

…There are also several private helpers to split up the work for spawning cabs (all tackling different parts of the cab spawn pseudo outlined in the spec). There are helpers for each of the different potential actor adders which are called in the move() function.

**Actor: public GraphObject)**

*Constructor:* Actors have a public constructor which initializes the GraphObject on which Actors are derived. In addition, all Actors are initialized with a vertical speed, horizontal speed, Boolean values indicated whether they are collision avoidance worthy and/or affected by projectiles, and a StudentWorld pointer linking them to their world. (Note: none of my constructors are virtual because that is impossible and doesn’t ever make sense).

*Destructor:* Actors have an empty virtual destructor. It is empty because there is no dynamic allocation. All of my destructors are virtual because it prevents improper destruction during polymorphism (which could occur if I later introduced dynamic allocation in one of my derived classes). **This reasoning holds true for all the rest of the destructors throughout this report.**

Actors also have the following public methods:

*Getters:*

*bool isCollisionAvoidanceWorthy() const:*

This simple getter method returns whether or not an Actor is a collision avoidance worthy actor. This is in the base class because all Actors are either collision avoidance worthy or not. This is not virtual because all actors can get this status in the same way.

*bool isAffectedByProjectiles() const:*

This getter returns whether or not an Actor is affected by projectiles. This is in the base class because all Actors are either affected by projectiles or not. This is not virtual because all actors can get this status in the same way.

*double getVertSpeed() const:*

This getter returns an actor’s vertical speed. This is in the base class because all Actors have a vertical speed (although not all actors use it in the current game implementation). This is not virtual because all actors can get this status in the same way.

*double getHorizSpeed() const:*

This getter returns an actor’s horizontal speed. This is in the base class because all Actors have a horizontal speed (although not all actors use it in the current game implementation). This is not virtual because all actors can retrieve their speed value in the same way.

*virtual bool isAlive() const = 0:*

This *pure virtual* getter returns whether an Actor is alive or not. I chose to make this function in the base Actor class because all actors have a life status, but pure virtual because each branch of derived classes treat their life status in a different way. For DamageableActors this is redefined such that life status is determined based on HP, and other actors simply have an on/off life status.

*Setters/Other:*

*void doSomething():*

All Actors can be called upon to doSomething(), therefore I defined this function in the base Actor class. Each branch of actors has a different pattern of doing something, but ALL actors first check whether they are alive before actually doing something. Therefore, I factored out this functionality (checking whether one is alive) in the abstract Actor base class. This factored out behavior does not vary between different kinds of actors so therefore doSomething() is not virtual. making sure the actor is alive this method calls a private *pure virtual* function called actuallyDoSomething(). That function is in the base Actor class because all actors can actuallyDoSomething(), but it is pure virtual because each kind of actor actually does something in different ways.

*void hitByProjectile(const int& amt):*

All Actors can technically be hitByProjectiles whether or not they are affected by them, so therefore I put this method in the base Actor class. It is not virtual because the functionality of this method is the same for all kinds of actors. This method first checks whether an actor is dead or is not affected by projectiles. If so, it returns. Otherwise, it calls a private virtual function called reactToProjectile(const int& /\* amt \*/). Details on that function described below.

*virtual void killActor() = 0:*

This method kills an actor. I put it in the base Actor class because all actors can be killed. I chose to make it pure virtual because killing an actor means different things depending whether they actor is damageable (has HP) or simply has an on/off based life status.

*Notable Protected/ Private Methods:*

*Bool moveRelativeToRacer():*

This function implements the movement algorithm that all actors other than GhostRacer use to move themselves relative to the racer’s speed (actual pseudo is outlined in the spec). This method is defined in the Actor base class and is not virtual because all actors do this in the same way. This method returns a Boolean value based on whether the object is still on the screen after moving so that actuallyDoSomething() functions can make sure to stop doing anything with objects that go off the screen.

*Bool hasExitedScreen():*

This function implements the basic check for whether an Actor has exited the screen as outlined in the spec and used frequently by most actors. It is defined in the base actor class since it has the same implementation for all actors, and it is not virtual because no actors check whether they’ve gone off screen in a different way.

*Virtual void reactToProjectile(const int& /\*amt\*/ ):*

This private virtual function by default calls the killActor() method, since most actors that are affected by projectiles instantly die when hit. However, this reactToProjectile(const int& ) method is virtual because Enemies react differently to projectiles and will go on to redefine the function. The function is in the base Actor class because a wide variety of Actors will reactToProjectiles.

*Virtual void actuallyDoSomething() = 0:*

This function is where the actual majority of code for each unique actor’s behavior when doing something will be implemented. I put this in the base Actor class because all actors can actuallyDoSomething() if alive, but it is marked pure virtual because each actor will have unique behavior (and in some cases classes in the middle of my inheritance tree will set up patterns of behavior in the actuallyDoSomething() implementation).

**Environment: public Actor)**

*Constructor:* The Environment Constructor initializes the Actor base class. It takes in all the same parameters as the Actor constructor except it doesn’t ask for CAW status. All Environment actors are NOT collision avoidance worthy, so the Environment constructor automatically passes false in for that parameter when initializing the base Actor class. Furthermore, all environment actors start out alive so m\_alive is initialized to true.

*Destructor:* Virtual empty destructor. (Just for good practice)

*Virtual bool isAlive() const:*

Environment actors redefine the isALive() status to simply return on/off life status. This method was redefined in the Environment class because all environment objects have the same on/off life status. Environment actors now have a private bool data member called m\_alive which indicates whether the actor is alive or not. The method is marked virtual because it is a redefinition of a pure virtual method from the base Actor class.

*Virtual void killActor():*

Killing an environment actor simply sets its m\_alive to false. This is defined in Environment because all Environment actors are killed this way. It is marked virtual because this is a redefinition of a pure virtual method from the base Actor class.

*Notable Protected/ Private Methods:*

n/a

**DamageableActor: public Actor)**

*Constructor:* DamageableActor’s constructor takes roughly the same parameters as the base class Actor except it doesn’t ask for CAW status and it asks additionally for HP. Instead of asking for CAW status, it passes in the value true for all damageable actors because all damageable actors are collision avoidance worthy. DamageableActors now have HP so m\_hitPoints is initialized to whatever HP is passed on for the specific kind of DamageableActor.

*Destructor:* Virtual empty destructor.

*Virtual bool isALive() const:*

DamageableActors define this method to return whether HP is greater than 0 or not. All damageable actors check their life status in this way so I defined the method in this DamageableActor class. This method is virtual because it is a redefinition of a pure virtual method from the base Actor class.

*int getHP() const:*

Returns the exact HP amount that a DamageableActor has. This getter is applicable to all damageable actors so I chose to implement it in this class. This method is not virtual because damageable actors access their HP in the same way.

*virtual void killActor():*

Sets DamageableActor HP to 0 (this is how damageable actors are killed since they have HP). This method is defined in the damageable actor class since all damageable actors are killed in this way. It is marked virtual because it is a redefinition of a pure virtual function from the base Actor class.

*void getDamaged(const int& amt):*

This method does nothing if a damageable actor is already dead. Otherwise it decrements the objects hit points by amt. If the object is dead after losing said hit points, then this method automatically calls the noHPLeft() method, a private virtual method which allows damageable actors to express themselves in their unique ways of dying. (Details below). This getDamaged() method is defined in the damageable actor class because all damageable actors get damaged with this same pattern of behavior. It is not virtual because the differences in behavior for getting damaged are factored out implicitly via different starting hit points for each damageable actor, and differently defined noHPLeft() methods.

*Notable Protected/ Private Methods:*

*Virtual void noHPLeft():*

This function implements how damageable actors act when killed. This function is virtual because several DamageableActors will redefine their ‘dying process’ behavior. By default it is empty because not all damageable actors will do anything at all when dying (or necessarily even have the capacity to die under the current overall game framework). I chose to put this function in the DamageableActors class because both GhostRacer and Enemies will go on to use this function (the two classes branching from DamageableActors).

**Goodie: public Environment)**

*Constructor:* Goodies have a constructor which takes all of the same parameters as the Environment constructor and passes them on to initialize the Environment base class.

*Destructor:* Virtual empty destructor (for good practice).

*Notable Protected/ Private Methods:*

*Virtual void actuallyDoSomething():*

All Goodies follow the following pattern of actually doing something:

Move relative to racer.

If still on the screen

If overlapping the racer and the racer is alive

interactWithRacer() (Details below)

extraMovements() (Details below)

Because of this common pattern of behavior amongst all goodies, I re-implement actuallyDoSomething() as above in this Goodie class. Furthermore, I mark it virtual because this is a re-implementation of a virtual function from the base Actor class.

*Virtual void interactWithRacer() = 0:*

This function implements each goodie’s specific way of interacting with Ghost Racer if they overlap with GhostRacer. This function is pure virtual because all goodies interact with the racer in unique ways, there is no default.

*Virtual void extraMovements():*

This function is empty by default in the Goodie class, because it is called by the Goodie pattern of actually doing something, but most goodies don’t have extra movements. Those that do (namely the soul goodie) implement their extra movements later, so this function is marked virtual.

**Enemy: public DamageableActor)**

*Constructor:* The Enemy constructor takes mostly the same parameters as DamageableActors but it does not ask for projectile affected status. Rather, it passes in the value true for all Enemies because all enemies are affected by projectiles. The Enemy constructor has one additional bool parameter called ‘zombieOrNot’ which is used to initialize a new private data member which keeps track of whether an enemy is a zombie or not. This data member is important because zombies are damaged by holy water, while the pesky human pedestrian enemies are not. Additionally, during construction all Enemies initialize their movement plan distance to 0 (another new data member).

*Destructor:* Virtual empty destructor (for good practice).

*Notable Protected/ Private Methods:*

*Virtual void actuallyDoSomething():*

All enemies follow the following pattern of actually doing something:

Try to mess with ghost racer (if overlapping him this will succeed)

If successful

Return

otherwise

move relative to the racer.

If still on the screen

Plan movements.

Because all enemies share this similar pattern of actually doing something, I defined this actually do something method in the Enemy class. It is virtual because it is a redefinition of the actuallyDoSomething() method from the base Actor class. Specifics on messing with ghost racer and planning movements are handled via other private methods implemented in each specific class.

*Void makeNewPedMovementPlan():*

Zombie peds and human peds both have the same method of making a new movement plan, so this functionality is factored out and zombie peds and human peds can call this protected method in their specific implementations. This is defined in the enemy class because it is useful for most enemies and prevents duplicate code. It is not virtual because the behavior is the same for each kind of pedestrian that uses it.

*Virtual void messWithGhostRacer() = 0:*

This function is a pure virtual function in the enemy class because all enemies can mess with ghost racer but each specific enemy defines its specific behavior in its class.

*Virtual void planMovements() = 0:*

This is a pure virtual function in the enemy class because all enemies can plan movements but each specific enemy does this in their own way (with some factored out behavior in protected methods).

*Virtual void reactToProjectile(const int& amt):*

This function is re-implemented in the enemy class because now enemies which are affected by projectiles can be damaged by the specific amt specified and they all share similar behavior. If an enemy is undead it will be damaged by a certain amount. Regardless of whether an enemy is a zombie or not, if it is alive after being hit, it will then perform some behavior defined in the private function enemyWasHitByProjectile() (details handled in the fully derived classes).

**GhostRacer: public DamageableActor)**

*Constructor:* GhostRacer’s constructor takes only one parameter: the StudentWorld pointer corresponding to the world that GhostRacer is being instantiated in. Everything else is always the same. The racer’s starting x, y, direction, size, image ID etc. are all laid out in the spec. This is all passed to DamageableActors’ constructor. GhostRacer now has an ammo count, in addition to everything else a DamageageableActor has and that is initialized to 10 during construction as per the spec.

*Destructor:* Virtual empty destructor (for good practice).

*Unsigned int getAmmo() const:*

GhostRacer has an added getter for her ammo count. This is defined in the GhostRacer class and is not virtual because only GhostRacer has ammo.

*Void getSpun():*

This method allows other objects (namely, the oil slicks) to spin Ghost Racer. It is defined in the GhostRacer class and is not virtual because only GhostRacer can be spun.

*Void getHealed(const int& amt):*

This method allows GhostRacer to be healed by the amt passed in (aka her HP is increased by amt). It is implemented in the GhostRacer class and not virtual because only the Ghost Racer can actually be healed.

*Void giveAmmo(const int& amt):*

This method allows GhostRacer to gain a certain amt of ammo (aka her ammo data member is increased by amt). It is implemented in the GhostRacer class and not virtual because only the Ghost Racer can gain ammo.

*Notable Protected/ Private Methods:*

*Virtual void actuallyDoSomething():*

GhostRacer has a unique implementation of actually doing something involving moving based on the user’s keystroke inputs (outlined in the spec). This is marked virtual because it is a redefinition of a base Actor class method and it is in the GhostRacer class because this specific implementation is unique to GhostRacer.

**HumanPedestrian: public Enemy)**

*Constructor:* The HumanPedestrian constructor takes parameters for x and y location, as well as a StudentWorld pointer. It passes constants as laid out in the spec for all other necessary parameters to construct the Enemy class on which HumanPedestrians are based. Humans are not undead and this is specified in addition to what the spec laid out (because of how my Enemy class is set up).

*Destructor:* virtual empty destructor (for good practice)

*Notable Protected/ Private Methods:*

Nothing much to note, but there are virtual re-implementations of messWithghostRacer(), planMovements(), and enemyWasHitByProjectile() to accomplish the specific behavior for this fully derived class based on the general pattern/ setup established in the Enemy class.

**ZombiePedestrian: public Enemy)**

*Constructor:* The ZombiePedestrian constructor takes parameters for x, y location as well as a StudentWorld pointer. It initializes the Enemy class on which ZombiePedestrians are based with these values as well as constant values for zombie peds as laid out in the spec. Additionally it passes in the fact that zombies ARE undead (true). The constructor also initializes a new data member in zombie pedestrians, m\_ticksUntilGrunt, with the value 0 (because all zombie peds start out with no ticks until their next grunt sound).

*Destructor:* virtual empty destructor (for good practice)

*Notable Protected/ Private Methods:*

Nothing much to note, but there are virtual re-implementations of messWithghostRacer(), planMovements(), and enemyWasHitByProjectile() to accomplish the specific behavior for this fully derived class based on the general pattern/ setup established in the Enemy class. There is also a virtual void noHPLeft() re-implementation because zombies die in a unique way.

**ZombieCab: public Enemy)**

*Constructor:* The ZombieCab constructor takes parameters for x, y locations as well as vertical speed and a StudentWorld pointer. It initializes the Enemy class on which ZombieCabs are based with these values as well as constant values for zombie cabs as laid out in the spec. Additionally it passes in the fact that zombie cabs ARE undead (true). The constructor also initializes a new data member in zombie cabs, m\_hasDamagedGhostRacer, which keeps track of whether the cab has already damaged the Ghost Racer. This is initialized to false.

*Destructor:* virtual empty destructor (for good practice)

*Notable Protected/ Private Methods:*

Nothing much to note, but there are virtual re-implementations of messWithghostRacer(), planMovements(), and enemyWasHitByProjectile() to accomplish the specific behavior for this fully derived class based on the general pattern/ setup established in the Enemy class. There is also a virtual void noHPLeft() re-implementation because zombie cabs die in a unique way.

**HolyWaterProjectile: public Environment)**

*Constructor:* The HolyWaterProjectile constructor takes x and y coordinates as well as direction and a StudentWorld pointer as parameters. It passes these, in addition to a bunch of constants defined in the spec describing how HolyWaterProjectiles should be initialized, to the Environment constructor. Additionally, the constructor initializes HolyWaterProjectile’s new m\_travelDistance data member to 160 as outlined in the spec.

*Destructor:* virtual empty destructor (for good practice).

*Notable Protected/ Private Methods:*

*Virtual void actuallyDoSomething():*

This is a virtual method because it re-implements a base Actor class method and it is set up in HolyWaterProjectile because its behavior is unique to this class. The projectile moves in a unique way and tries to maybe damage/hit an actor if a projectile affected actor happens to overlap with it. (Details in spec).

**BorderLine: public Environment)**

*Constructor:* The BorderLine constructor has an imageID passed in (to distinguish between white and yellow borders), as well as x and y location, and a StudentWorld pointer. These parameters, in addition to constants laid out in the spec, are passed into the Environment constructor to construct the base Environment class on which BorderLines are based.

*Destructor:* virtual empty destructor (for good practice)

*Notable Protected/ Private Methods:*

Not much to note, but BorderLine has the simplest re-implementation of actuallyDoSomething() which simply moves it relative to the Ghost Racer. And of course it kills itself if off screen (but that is handled by the moveRelativeToRacer() function).

**OilSlick: public Goodie)**

*Constructor:* The OilSlick constructor takes in x and y location as well as a StudentWorld pointer for parameters. It then calls the constructor for the Goodie class on which oil slicks are derived with these passed values and additionally a number of constants as laid out in the spec.

*Destructor:* virtual empty destructor (for good practice).

*Notable Protected/ Private Methods:*

Oil Slick redefines its interaction with racer to handle its unique behavior when overlapping the racer. (All goodies redefine this).

**HealingGoodie: public Goodie)**

*Constructor:* The HealingGoodie constructor takes x location, y location, and a StudentWorld pointer as parameters. It then calls the Goodie constructor with these and other constants that are laid out in the spec.

*Destructor:* virtual empty destructor (for good practice).

*Notable Protected/ Private Methods:*

Same as above.

**HolyWaterGoodie: public Goodie)**

*Constructor:* The HolyWaterGoodie constructor takes in x, y location and StudentWorld pointer and then calls the Goodie constructor with these and other constants that are laid out in the spec.

*Destructor:* virtual empty (for good practice).

*Notable Protected/ Private Methods:*

Same as above.

**SoulGoodie: public Goodie)**

*Constructor:* The SoulGoodie constructor takes in x, y location and StudentWorld pointer and then calls the Goodie constructor with these and other constants laid out for SoulGoodies in the spec.

*Destructor:* virtual empty (for good practice).

*Notable Protected/ Private Methods:*

Same as above, but also virtual void extraMovements() is re-implemented since Soul Goodies spin (virtual b/c re-implementation, and in Soul Goodie class because unique to Soul Goodies).

**Missing Functionality/ Bugs:**

I do not have any known functionality missing, nor am I aware of any bugs in my classes.

**Other Design Decisions/ Assumptions:**

Although it was specified that the racer should never spin past 60 or 120 degrees, it was not specified what exactly the getSpun() method should do when an oil slick attempts to spin the racer past those values. I assumed that the racer should still spin, but just right up to either 60 or 120 degrees (rather than not spinning in that tick at all).

For adding zombie pedestrians and human pedestrians I added them between 0 and VIEW\_WIDTH-1 (aka on the screen). The spec was unclear about this but I discussed this decision with the profs.

I assumed that it would be considered cleaner to have constants in classes rather than hard-coded numbers for constant constructor values (and therefore used constants wherever possible).

For checking whether actors have exited the screen I checked whether X went below 0 or greater than VIEW\_WIDTH or if Y went below 0 or greater than VIEW\_HEIGHT. This did not seem logical given that VIEW\_WIDTH and VIEW\_HEIGHT are both off the screen by 1 pixel, but I just followed what the spec said.

It wasn’t explicitly noted in the spec, but I assumed that the HolyWaterProjectile should decrement its travel distance by SPRITE\_HEIGHT after each time that it moves forward. This produced the same behavior that I saw while playing the sample game.