**Descriptions of public member functions**

**Actor**

**virtual** **bool** collisionAvoidanceWorthy() { **return** **true**; }

* I put this function in the Actor base class because all Actors have property of either being collisionAvoidanceWorthy or not
* I made this function virtual because most Actors have this property, but some Actors do not (in which case, they can override this function)

**virtual** ~Actor() {}

* I put this function in the Actor base class because all Actors have a destructor
* The Actor destructor must be virtual so that the subclasses can also have their own destructors

**void** setToDead() { m\_isAlive = **false**; }

* I put this function in the Actor base class because all actors have the property of alive-ness
* I made this function non-virtual because its implementation is the same for all Actors

**double** getVertSpeed() { **return** m\_vertSpeed; }

* I put this function in the Actor base class because all actors have the property of vertical speed
* I made this function non-virtual because its implementation is the same for all Actors

**virtual** **void** receiveDamage(**int** damage) {}

* I put this function in the Actor base class because
* I made this function non-virtual because its implementation is the same for all Actors

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

* I put this function in the Actor base class because all Actors doSomething()
* I made this function pure virtual because its implementation is different for all Actors directly derived from Actor

**bool** isAlive() { **return** m\_isAlive; }

* I put this function in the Actor base class because all Actors have the property of alive-ness
* I made this function non-virtual because its implementation is the same for all Actors

**int** getLaneNum();

* I put this function in the Actor base class because all Actors have a lane number
* I made this function non-virtual because its implementation is the same for all Actors

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

* I put this function in the Actor base class because all Actors are affectedByProjectiles
* I made this function pure virtual because each Actor must be affectedByProjectiles or not

**bool** doOverlap(Actor\* otherActor);

* I put this function in the Actor base class because all Actors can overlap
* I made this function non-virtual because its implementation is the same for all Actors

**Active**

**void** addHealth(**int** health);

* I put this function in the Active base class because all Active Actors have health
* I made this function non-virtual because all Active Actors treat health in the same way

**int** getHealth();

* I put this function in the Active base class because all Active Actors have health
* I made this function non-virtual because all Active Actors treat health in the same way

**virtual** **bool** isAffectedProjectiles() { **return** **true**; }

* I put this function in the Active base class because all Active Actors must have this property
* I made this function virtual because almost all Active Actors have this property as true

**bool** collisionAvoidanceWorthy() { **return** **true**; }

* I put this function in the Active base class because all Active Actors must have this property
* I made this function non-virtual because all Active Actors have this property as true

**virtual** ~Active() {}

* I put this function in the Active base class because all Active Actors have a destructor
* The Active destructor must be virtual so that the subclasses can also have their own destructors

**Goodie**

**virtual** **bool** isAffectedProjectiles() { **return** **false**; }

* I put this function in the Goodie base class because all Goodie Actors must have this property
* I made this function virtual because most Goodie Actors are affected by projectiles but some are not

**virtual** **bool** collisionAvoidanceWorthy() { **return** **false**; }

* I put this function in the Goodie base class because all Goodie Actors must have this property
* I made this function virtual because even though all Goodie Actors are not collisionAvoidanceWorthy, it's good style to make it virtual since I'm inheriting it from a base class

**virtual** ~Goodie() {}

* I put this function in the Goodie base class because all Goodie Actors have a destructor
* The Goodie destructor must be virtual so that the subclasses can also have their own destructors

void receiveDamage(int damage) { setToDead(); }

* I put this function in the Goodie base class because all Goodie Actors that are affected by holy water projectiles are damaged by them (and only damaged by holy water projectiles)
* The function is non-virtual because all Goodie Actors that are affected by holy water projectiles are damaged by them in the same way

**Oil Slick**

**virtual** ~OilSlick() {}

* I put this function in the Oil Slick class because it needs to be there for inheritance purposes
* The Oil Slick destructor must be virtual since it inherits a destructor from a superclass

**Healing Goodie**

**virtual** ~HealingGoodie() {}

* I put this function in the Healing Goodie class because it needs to be there for inheritance purposes
* The Healing Goodie destructor must be virtual since it inherits a destructor from a superclass

**virtual** **bool** isAffectedProjectiles() { **return** **true**; }

* I put this function in the Healing Goodie class because Healing Goodies, unlike most others Goodie Actors, are affected by projectiles
* I made this function virtual because it's good style to make it virtual, since it inherits it from the Goodie class

**Holy Water Goodie**

**virtual** ~HealingGoodie() {}

* I put this function in the Holy Water Goodie class because it needs to be there for inheritance purposes
* The Holy Water Goodie destructor must be virtual since it inherits a destructor from a superclass

**Soul Goodie**

**virtual** ~SoulGoodie() {}

* I put this function in the Soul Goodie class because it needs to be there for inheritance purposes
* The Soul Goodie destructor must be virtual since it inherits a destructor from a superclass

**Borderline**

**virtual** ~BorderLine() {}

* I put this function in the Borderline class because it needs to be there for inheritance purposes
* The Borderline destructor must be virtual since it inherits a destructor from a superclass

**Holy Water Projectile**

**virtual** ~HolyWaterProjectile() {}

* I put this function in the HolyWaterProjectile class because it needs to be there for inheritance purposes
* The HolyWaterProjectile destructor must be virtual since it inherits a destructor from a superclass

**Pedestrian**

**virtual** ~Pedestrian() {}

* I put this function in the Pedestrian class because it needs to be there for inheritance purposes
* The Pedestrian destructor must be virtual since it inherits a destructor from a superclass

**Human Pedestrian**

**virtual** ~HumanPedestrian() {}

* I put this function in the Human Pedestrian class because it needs to be there for inheritance purposes
* The HumanPedestrian destructor must be virtual since it inherits a destructor from a superclass

**Zombie Pedestrian**

**virtual** ~ZombiePedestrian() {}

* I put this function in the Zombie Pedestrian class because it needs to be there for inheritance purposes
* The ZombiePedestrian destructor must be virtual since it inherits a destructor from a superclass

**Zombie Cab**

**virtual** ~ZombiePedestrian() {}

* I put this function in the Zombie Cab class because it needs to be there for inheritance purposes
* The ZombieCab destructor must be virtual since it inherits a destructor from a superclass

**Ghost Racer**

**virtual** ~GhostRacer() {} //virtual functions must be defined (Even if they're empty)

* I put this function in the GhostRacer class because it needs to be there for inheritance purposes
* The GhostRacer destructor must be virtual since it inherits a destructor from a superclass

**virtual** **bool** isAffectedProjectiles() { **return** **false**; }

* I put this function in the GhostRacer class because unlike most Active classes, it is not affected by projectiles
* I made this function virtual since it inherits it from a superclass

**virtual** **void** receiveDamage(**int** damage);

* I put this function in the GhostRacer class it has a different implementation than all other classes
* I made this function virtual since it inherits it from a superclass

**void** addWater(**int** charge);

* I put this function in the GhostRacer class because it is not found in any other class
* I made this function non-virtual since it is not found in any other class

**void** spin();

* I put this function in the GhostRacer class because it is not found in any other class
* I made this function non-virtual since it is not found in any other class

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

* I put this function in the GhostRacer class because it is implemented very differently from all other classes
* I made this function virtual since it is inherited from a superclass

**int** getSprays();

* I put this function in the GhostRacer class because it is not found in any other class
* I made this function non-virtual since it is not found in any other class

**StudentWorld**

**virtual** ~StudentWorld();

* I put this function in the StudentWorld class because it needs to be there for inheritance purposes
* The StudentWorld destructor must be virtual since it inherits a destructor from a superclass

GhostRacer\* getGhostRacer();

* I put this function in the StudentWorld class because many Actors need to access it
* I made this function non-virtual since it is only implemented in this class

Actor\* closestInLane(**int** laneNum, **double** y);

* I put this function in the StudentWorld class because some Actors need to access it
* I made this function non-virtual since it is only implemented in this class

Actor\* getProjectileOverlap(HolyWaterProjectile\* proj);

* I put this function in the StudentWorld class because some Actors need to access it
* I made this function non-virtual since it is only implemented in this class

**void** addHealingGoodie(**double** startX, **double** startY);

* I put this function in the StudentWorld class because some Actors need to access it
* I made this function non-virtual since it is only implemented in this class

**void** addOilSlick(**double** startX, **double** startY);

* I put this function in the StudentWorld class because some Actors need to access it
* I made this function non-virtual since it is only implemented in this class

**void** addHolyWaterSpray(**double** startX, **double** startY, **int** startDirection);

* I put this function in the StudentWorld class because some Actors need to access it
* I made this function non-virtual since it is only implemented in this class

**void** incNumSaved();

* I put this function in the StudentWorld class because some Actors need to access it
* I made this function non-virtual since it is only implemented in this class

**Functionality I failed to finish**

* None

**Known bugs**

* The Zombie Cabs collide through each other.

**Assumptions**

* Whenever the spec says "make a new actor at y location of view\_height," it's actually view\_height-1, right? Since the bounds of the game are [0, view\_height)
* The implementation of the movement plan distance for the Zombie Pedestrian and the Zombie Cab are effectively the same, though they do differ in the order that you decrement.
* I assumed that we could move init(), move(), and cleanup() from the public section to the private section of StudentWorld, since that’s good OOP coding style.