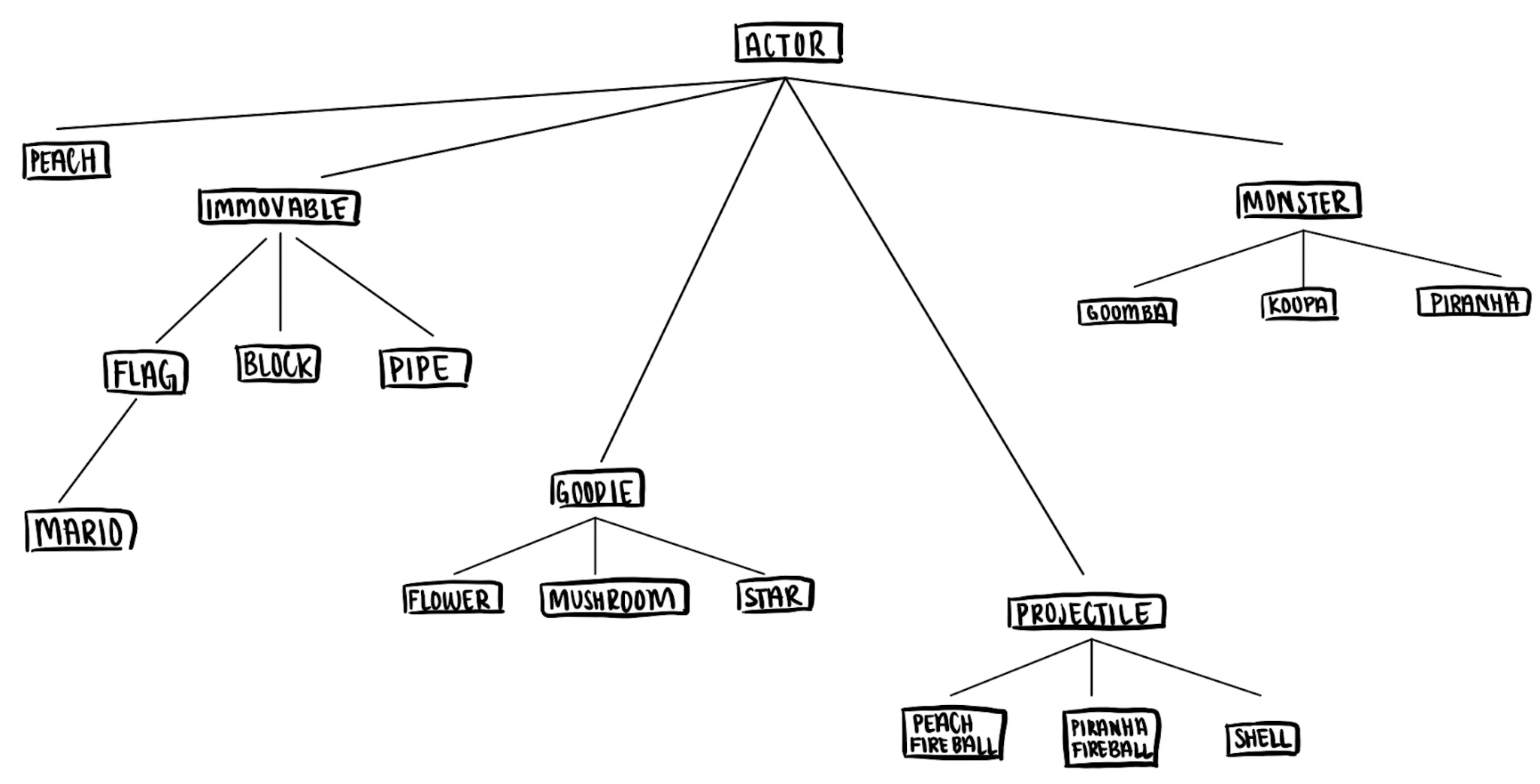
Gabriella Ang

Carey Nachenberg

Computer Science 32

**Project 3 Report**



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

**StudentWorld Class**

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

*StudentWorld(string assetPath);*

This is the constructor for the StudentWorld class which is derived from the GameWorld object.

*virtual ~StudentWorld();*

This is the destructor for the StudentWorld class. I made it virtual because it’s derived from the GameWorld class and so that it destroys everything from all of it’s classes. It calls the cleanUp() function.

*virtual int init();*

init() is a mandatory function and it initializes all the objects/actors for every level. It was made a virtual function because it is derived from the GameWorld class.

*virtual int move();*

move() was a mandatory function that tells each of the objects that were created to do something as well as . It was made a virtual function because it is derived from the GameWorld class.

*virtual void cleanUp();*

cleanUp() is a mandatory function that deletes all of the objects that were initialized in the game. It was created to be a virtual function because it is derived from the GameWorld class.

*bool overlap(int x, int y, Actor\* &character);*

This function checks to see if there is an overlap with an Actor at a given (x, y) coordinate. It returns true if it detects an overlap and false otherwise. It also takes a reference &character that gives the reference to the Actor it overlaps with if it returns true. I chose to define it in the StudentWorld class because it requires the actors vector. This is so that I can perform actions on specific Actors that are in the actors vector.

*bool overlap(Actor \* a, Actor \* b);*

This function checks the overlap between two Actors. Though it is very similar to the first overlap function, I decided to overload the overlap function because I wanted to be able to check for overlap without necessarily knowing the specific position of the objects. If I had had more time to work on this project, I would probably have figured out a way to consolidate the two functions into one. I chose to define it in the StudentWorld class because it requires the actors vector. This function was only used by the StudentWorld class so I did not include the virtual keyword.

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

This function checks if there is an object that would return true to the blocksMovement() function and returns true if there is indeed an object that blocks movement at the given (x, y) coordinate. I chose to define it in the StudentWorld class because it requires the actors vector. This function was only used by the StudentWorld class so I did not include the virtual keyword.

*bool damagableObject(int x, int y, Actor\* &character);*

This function checks if there is an object that would return true to the isDamagable() function and returns true if indeed there is indeed an object that can be damaged at the given (x, y) coordinate. Additionally, if it returns true, it also gives the object that can be damaged in the character reference variable. I chose to define it in the StudentWorld class because it requires the actors vector. This function was only used by the StudentWorld class so I did not include the virtual keyword.

*void setStatus(int status);*

This function simply sets the status of the StudentWorld. I chose to define it in the StudentWorld class because it defines the status of the whole game. This function was only used by the StudentWorld class so I did not include the virtual keyword.

*int getStatus();*

This function simply returns the status of StudentWorld. This function was only used by the StudentWorld class so I did not include the virtual keyword.

*Peach\* getPeach();*

This function simply returns the Peach object. This function was only used by the StudentWorld class so I did not include the virtual keyword.

*void addActor(Actor\* a);*

This function adds an actor object to the actors vector. This function was only used by the StudentWorld class so I did not include the virtual keyword.

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

**Actor Class**

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

*Actor(StudentWorld \* sw, int x, int y, int ID, int depth, bool state);*

This is the base class constructor for all the actors. It was derived from the GraphObject class.

*virtual void doSomething() = 0;*

This doSomething() function will be inherited to all the derived classes defined later. I chose to put it in the Actor class because all actors have a doSomething() function. I also chose to make it a pure virtual function because I will never call on the Actor class so it made more sense to make it an Abstract Base Class.

*int getImageID();*

This function simply returns the image ID of any actors. I chose to define it in the Actor class because all actors have an imageID. I chose not to define it as a virtual or pure virtual function because this function stays the same in all the derived classes.

*int getDepth();*

This function simply returns the depth of any actors. I chose to define it in the Actor class because all actors have a depth. I chose not to define it as a virtual or pure virtual function because this function stays the same in all the derived classes.

*bool isAlive();*

This function simply returns the state (alive/dead) of any actor. I chose to define it in the Actor class because all actors have a state. I chose not to define it as a virtual or pure virtual function because this function stays the same in all the derived classes.

*StudentWorld\* getWorld();*

This function returns the StudentWorld object that it is in. I chose to define it in the Actor class because all or most of the actors need to call on the StudentWorld it is in at some point. I chose not to define it as a virtual or pure virtual function because this function stays the same in all the derived classes.

*virtual void bonk() = 0;*

The bonk() function is for whenever any of the Actor objects are “bonked”. I chose to define it in the Actor class because all derived objects can be bonked. I also chose to make it a pure virtual function because the Actor class will never be called on. The actual Actor class does not have a specific defined bonk and thus will never need to be called on.

*virtual bool isDamagable() = 0;*

The isDamagable() function simply returns a boolean value for whether or not the object is able to be damaged/killed. I choose to put it in the Actor class because this is a function that all actors use. I also chose to make it a pure virtual function because the Actor class does not specifically have a true/false value because it is very general.

*virtual bool blocksMovement() = 0;*

The blocksMovement() function simply returns a boolean value for whether or not the object is able to block the movement of Peach or other objects. I choose to put it in the Actor class because this is a function that all actors use. I also chose to make it a pure virtual function because the Actor class does not specifically have a true/false value because it is very general.

*void setState(bool state);*

This function simply sets the state of the actor to alive/dead (true/false). I put it in the Actor base class because all/most of the Actors can use this function. I chose not to define it as a virtual or pure virtual function because this function stays the same in all the derived classes.

*bool getState();*

This function simply returns the state of an object. I put it in the Actor base class because all/most of the Actors can use this function. I chose not to define it as a virtual or pure virtual function because this function stays the same in all the derived classes.

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

**Peach Class**

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

*Peach(StudentWorld \* sw, int x, int y);*

This is the Peach object constructor derived from the Actor class.

*virtual void doSomething();*

This is Peach’s doSomething() function specific to her (which is why it’s defined in this class). In this class, Peach moves (according to the key pressed), shoots/creates PeachFireball objects if she is able, as well as checks for damage or damages/bonks other objects. This function is defined as virtual because it derives itself from the Actor class.

*virtual void bonk();*

The bonk() function is used to determine what happens to Peach when she gets hit by a monster or fireball. If she has starPower or temporary invincibility, she bonks the enemy instead, but if not, she dies. I chose to put this function in the Peach class because bonk() is very specific to Peach and her bonk() is different from other Actor objects. Additionally, I defined it as a virtual function because it is derived from the Actor object.

*virtual bool isDamagable();*

The isDamagable() function simply returns a boolean value for whether or not the object is able to be damaged/killed. This redefines the pure virtual function from the Actor object, hence why it is in the Peach class and why it is a virtual function.

*virtual bool blocksMovement();*

The blocksMovement() function simply returns a boolean value for whether or not the object is able to block the movement of Peach or other objects. This redefines the pure virtual function from the Actor object, hence why it is in the Peach class and why it is a virtual function.

*void setStarPower(int ticks);*

This just sets m\_starPower = 10 if Peach gets star power. This function is specific to the Peach Actor and thus is not virtual or pure virtual.

*void setShootPower(bool shoot);*

This just sets m\_shootPower to true if Peach gets shoot power and increases Peach’s hit points. This function is specific to the Peach Actor and thus is not virtual or pure virtual.

*void setJumpPower(bool jump);*

This just sets m\_jumpPower to true if Peach gets jump power and increases Peach’s hit points. This function is specific to the Peach Actor and thus is not virtual or pure virtual.

*bool getStarPower();*

This function returns whether or not Peach has star power. This function is specific to the Peach Actor and thus is not virtual or pure virtual.

*bool getJumpPower();*

This function returns whether or not Peach has jump power. This function is specific to the Peach Actor and thus is not virtual or pure virtual.

*bool getShootPower();*

This function returns whether or not Peach has shoot power. This function is specific to the Peach Actor and thus is not virtual or pure virtual.

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

**Immovable Class**

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

*Immovable(StudentWorld \* sw, int x, int y, int ID, int depth);*

This is the constructor for the Immovable class which is derived from the Actor class. It also has child functions: Block, Pipe, Flag, and Mario.

*virtual void doSomething();*

This function basically just returns when it is prompted to doSomething() by the StudentWorld object. This is because the immovable objects usually don’t do anything for the ticks. I choose to put it in the Immovable class instead of the derived classes because it basically does the same thing for most of the classes. If it doesn’t however, I made the function virtual so that it can be redefined.

*virtual void bonk();*

All this function does is play the bonk sound. I choose to put it in the Immovable class instead of the derived classes because it basically does the same thing for most of the classes. If it doesn’t however, I made the function virtual so that it can be redefined.

*virtual bool isDamagable();*

The isDamagable() function simply returns a boolean value for whether or not the object is able to be damaged/killed and sets it to false. This redefines the pure virtual function from the Actor object, hence why it is in the Immovable class and why it is a virtual function.

*virtual bool blocksMovement();*

The blocksMovement() function simply returns a boolean value for whether or not the object is able to block the movement of Peach or other objects and it returns true. This redefines the pure virtual function from the Actor object, hence why it is in the Immovable class and why it is a virtual function.

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

**Block Class**

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

*Block(StudentWorld \* sw, int x, int y, int goodie = -1);*

This is the constructor for the Block class that was derived from the Immovable class. The only difference between this and the Immovable class is that it may or may not have goodies.

*virtual void bonk();*

This function releases a goodie if it has one or just plays the bonk sound if it doesn’t. I put it in the Block class because it is specific to the block class. It is a virtual function because it is redefining the bonk() function in the Immovable class.

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

**Pipe Class**

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

*Pipe(StudentWorld \* sw, int x, int y);*

This is the constructor for the Pipe class which is derived from the Immovable class. It is basically the exact same as the Immovable class but it has a different image ID which is why I put it in it’s own class. If I had more time, I probably would’ve taken this class out and worked just with the Immovable class for this function.

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

**Flag Class**

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

*Flag(StudentWorld \* sw, int x, int y, int ID = IID\_FLAG);*

This is the constructor for the Flag class which is derived from the Immovable class. It is also the parent class of the Mario class.

*virtual void doSomething();*

This function basically just checks if it overlaps with Peach and if it does, finishes the level and moves on. I put it in the Block class because it is specific to the block class. It is a virtual function because it is redefining the bonk() function in the Immovable class.

*virtual bool blocksMovement();*

The blocksMovement() function simply returns a boolean value for whether or not the object is able to block the movement of Peach or other objects and it returns false. This redefines the virtual function from the Immovable object, hence why it is in the Flag class and why it is a virtual function.

*virtual bool isLastLevel();*

This function just returns a boolean of whether it is the last level or not and it returns false. This is for the doSomething() function so that it can check for the last level and can act correspondingly. This function is in the flag class because it’s the goal object and thus is only a characteristic of this class and the one that is derived from it. I also made it a virtual function because the Mario class derived from it will change the value.

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

**Mario Class**

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

*Mario(StudentWorld \* sw, int x, int y);*

This is the constructor for the Mario class which is derived from the Flag class.

*virtual bool isLastLevel();*

The only difference between this and the Flag class is that this function returns true which is why it is in this class. This is a virtual function because it is redefining the function from the Flag class.

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

**Goodies Class**

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

*Goodies(StudentWorld \* sw, int x, int y, int ID);*

This is the constructor for the Goodies class which is derived from the Actor class. It also has three child classes: Star, Mushroom, and Flower.

*virtual void doSomething();*

This function checks if it’s overlapping with Peach and then set’s her respective power according to the ID of the Goodie. This is in the Goodies class because it is similar in all the derived Goodies classes. This is defining the doSomething() pure virtual function from the Actor class which is why it is labeled as virtual.

*virtual bool isDamagable();*

The isDamagable() function simply returns a boolean value for whether or not the object is able to be damaged/killed and sets it to false. This redefines the pure virtual function from the Actor object, hence why it is in the Immovable class and why it is a virtual function.

*virtual bool blocksMovement();*

The blocksMovement() function simply returns a boolean value for whether or not the object is able to block the movement of Peach or other objects and it returns false. This defines the pure virtual function from the Actor object, hence why it is in the Goodies class and why it is a virtual function.

*virtual void bonk();*

This function just returns because the Goodies can’t be “bonked”. This is the same for all the derived Goodies which is why it’s in this class. This is a virtual function because it is defining the bonk() pure virtual function from the Actor class.

*int getPointValue();*

This function simply returns the amount of points it can give Peach when it overlaps with Peach. This is something that is for all Goodies so that’s why it’s in this class. It’s not a virtual or pure virtual function because it never needs to be redefined because it’s the same for all derived classes.

*int getPower();*

This function returns the number corresponding to each of the Goodies. It’s not a virtual or pure virtual function because it never needs to be redefined because it’s the same for all derived classes and is why it’s in the Goodie class.

*void setPointValue(int points);*

This function simply sets the point value of the Goodie or how many points to give Peach if she overlaps with this Goodie. It’s not a virtual or pure virtual function because it never needs to be redefined because it’s the same for all derived classes and is why it’s in the Goodie class.

*void setPower(int power);*

This function simply returns the point value of the Goodie or how many points to give Peach if she overlaps with this Goodie. It’s not a virtual or pure virtual function because it never needs to be redefined because it’s the same for all derived classes and is why it’s in the Goodie class.

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

**Flower Class**

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

*Flower(StudentWorld \* sw, int x, int y);*

This is the constructor for the Flower class which is derived from the Goodies class. Had I had more time to revise my code, I probably would have consolidated this into the Goodies class because there’s not really a need for this class.

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

**Mushroom Class**

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

*Mushroom(StudentWorld \* sw, int x, int y);*

This is the constructor for the Mushroom class which is derived from the Goodies class. Had I had more time to revise my code, I probably would have consolidated this into the Goodies class because there’s not really a need for this class.

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

**Star Class**

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

*Star(StudentWorld \* sw, int x, int y);*

This is the constructor for the Star class which is derived from the Goodies class. Had I had time to revise my code, I probably would have consolidated this into the Goodies class because there’s not really a need for this class.

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

**Projectile Class**

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

*Projectile(StudentWorld \* sw, int x, int y, int ID, int dir);*

This is the constructor for the Projectile class which is derived from the Actor class. This class is also the parent function of the PeachFireball, PiranhaFireball, and Shell classes.

*virtual void doSomething();*

The doSomething() function basically moves and bonks objects if they are able to be bonked. This function is in the Projectile class because it is derived from the Actor class and it needs to define this pure virtual function (which is also why it’s a virtual function).

*virtual bool isDamagable();*

The isDamagable() function simply returns a boolean value for whether or not the object is able to be damaged/killed and sets it to false. This redefines the pure virtual function from the Actor object, hence why it is in the Projectile class and why it is a virtual function.

*virtual bool blocksMovement();*

The blocksMovement() function simply returns a boolean value for whether or not the object is able to block the movement of Peach or other objects and it returns false. This defines the pure virtual function from the Actor object, hence why it is in the Projectile class and why it is a virtual function.

*virtual void bonk();*

This function just returns because projectile’s can’t really bonk anything. It’s in this class because Projectile is derived from Actor and it needs to define Actor’s pure virtual function bonk() (which is also why it’s a virtual function).

*virtual bool causeDamage() = 0;*

This function basically checks whether or not it causes damage and returns a boolean value. I put it in this class because all of the derived classes have different ways of causing damage and because this general Projectile class doesn’t have a specified way of causing damage. It’s also a pure virtual function for this reason because I wanted all the derived classes to implement this function specific to them.

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

**PiranhaFireball Class**

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

*PiranhaFireball(StudentWorld \* sw, int x, int y, int dir);*

This is the constructor for the PiranhaFireball class which is derived from the Projectile class.

*virtual bool causeDamage();*

This function checks if it overlaps with Peach and bonks her if it does. It returns a boolean value on whether or not it did bonk Peach or not. This function is in the PiranhaFireball class because it is specific to the PiranhaFireball and not to any other Projectile. This is a virtual function because it is derived from the Projectile class which has causeDamage() as a pure virtual function so I had to define the function.

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

**PeachFireball Class**

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

*PeachFireball(StudentWorld \* sw, int x, int y, int dir);*

This is the constructor for the PeachFireball class which is derived from the Projectile class.

*virtual bool causeDamage();*

This function basically just checks if it overlaps with an actor that is able to be damaged and returns true if it does and false otherwise. It also moves differently from the other Projectile-derived classes. This function is in the PeachFireball class because it is specific to the PeachFireball and not to any other Projectile. This is a virtual function because it is derived from the Projectile class which has causeDamage() as a pure virtual function so I had to define the function.

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

**Shell Class**

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

*Shell(StudentWorld \* sw, int x, int y, int dir);*

This is the constructor for the Shell class which is derived from the Projectile class.

*virtual bool causeDamage();*

This function basically just checks if it overlaps with an actor that is able to be damaged and returns true if it does and false otherwise. It also moves differently from the other Projectile-derived classes. This function is in the PeachFireball class because it is specific to the PeachFireball and not to any other Projectile. This is a virtual function because it is derived from the Projectile class which has causeDamage() as a pure virtual function so I had to define the function.

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

**Monster Class**

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

*Monster(StudentWorld \* sw, int x, int y, int ID);*

This is the constructor for the Monster class which is derived from the Actor class. It is also the parent class of the Goomba, Koopa, and Piranha class.

*virtual void doSomething();*

The doSomething() function basically moves and bonks Peach if they overlap with her. This function is in the Monster class because it is derived from the Actor class and it needs to define this pure virtual function. It is also defined as virtual because the derived classes have some specificity that require this function to be changed.

*virtual void bonk();*

This function basically sets the state of this Monster to false and increases the points. It is in this class because it does the same thing for all its derived classes. Additionally, it is defined as virtual because it is derived from the pure virtual bonk() function in the Actor class.

*virtual bool isDamagable();*

The isDamagable() function simply returns a boolean value for whether or not the object is able to be damaged/killed and sets it to true. This redefines the pure virtual function from the Actor object, hence why it is in the Monster class and why it is a virtual function.

*virtual bool blocksMovement();*

The blocksMovement() function simply returns a boolean value for whether or not the object is able to block the movement of Peach or other objects and it returns false. This defines the pure virtual function from the Actor object, hence why it is in the Monster class and why it is a virtual function.

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

**Goomba Class**

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

*Goomba(StudentWorld \* sw, int x, int y);*

This is the constructor for the Goomba class which is derived from the Monster class. If I had more time, I probably would have changed this to be my monster class and have the rest of the Monster-derived classes derive from this class.

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

**Koopa Class**

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

*Koopa(StudentWorld \* sw, int x, int y);*

This is the constructor for the Koopa class which is derived from the Monster class.

*virtual void bonk();*

This bonk() function adds a new shell object when it is bonked by Peach and then sets its state to false. It is in this class because it is different from the bonk of the other Monster-derived classes. It’s a virtual function because it is redefining the bonk() function from the Monster class.

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

**Piranha Class**

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

*Piranha(StudentWorld \* sw, int x, int y);*

This is the constructor for the Piranha class which is derived from the Monster class.

*virtual void doSomething();*

This function checks Peach’s position and determines whether or not to create a new PiranhaFireball object. This function is in this class because the Piranha class is redefining the doSomething() function from the Monster class. It’s a virtual function for the same reason.

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

**Incomplete Functionality**

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

I don’t think there is anything I missed when doing this project, however, there may have been things I missed that caused bugs I am unaware about.

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

**Design Decisions and Assumptions**

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

The only assumption I think I made was one about the text at the top of the screen. The spec did not specify anything about how the points were supposed to be formatted, but after playing the sample, I saw that it took up six digits and was filled with zeros if there was not a number in that position. Thus, I used a string stream to fill the spaces.