UID: 005738216

**SuperPeachSisters Report**

**Actor Class:**

I chose to define the functions doSomething(), getWorld(), isAlive(), hinderMvmt(), dead(), bonk(), damageable(), movement(), and damage() in the Actor class as they are related to all actors in the game.

* Actor(int ID, int x, int y, int dir, int depth, double size, StudentWorld\* w, bool hinder, bool damage);
  + Constructs an Actor object and is neither virtual nor pure virtual since it is a constructor
* virtual ~Actor()
  + Virtual destructor for the Actor class. It is empty, but is included because there is inheritance and polymorphism
* virtual void doSomething() = 0;
  + I defined this function as pure virtual since each Actor must do something during a tick and it differs for different actors. The doSomething function of each derived class is responsible for controlling the actor’s behaviour and action during a single tick of the game.
* StudentWorld\* getWorld();
  + Afunction which returns a pointer to StudentWorld so that Actors have the ability to access the world they are in. This function wasn’t made virtual nor pure virtual since it returns the same thing for all Actors.
* bool isAlive() const;
  + This function is const as it doesn’t change anything, but solely returns a boolean value as to whether an actor is alive or not. This function isn’t virtual nor pure virtual since it returns the value held by a private member variable in Actor which cannot be accessed by any derived classes.
* bool hinderMvmt() const;
  + This function is const as it doesn’t change anything, but solely returns a boolean value as to whether an actor blocks the movement of other actors. This function isn’t virtual nor pure virtual since it returns the value held by a private member variable in Actor which cannot be accessed by any derived classes.
* bool damageable() const;
  + This function is const as it doesn’t change anything, but solely returns a boolean value as to whether an actor can be damaged. This function isn’t virtual nor pure virtual since it returns the value held by a private member variable in Actor which cannot be accessed by any derived classes.
* virtual void dead();
  + This function is virtual. This function doesn’t return a value but is called by an actor when it dies. I defined it as virtual since all actors can die, but the action performed by an actor when it dies may differ. However, since many actors have die in the same way, I didn’t make it pure virtual, but instead implemented it in the Actor class as a function that simply sets the actor’s m\_alive variable to false.
* virtual void bonk();
  + This function is virtual. This function doesn’t return a value, but is called by an actor when it is bonked by another actor. I defined it as virtual since while some actors don’t do anything when bonked, other actors perform different actions wen bonked. For example, goodies don’t do anything when bonked, but other Actors such as Goombas and Koopas do perform an action when bonked.
* virtual void damage();
  + This function is virtual. This function doesn’t return a value but is called by an actor when it is damaged by another actor. I defined it as virtual since while you can attempt to damage any actor, not all actors will be damaged. For example, blocks cannot be damaged, but Peach can be damaged.
* int movement();
  + This function is neither virtual nor pure virtual. I made this function since some Actors such as projectiles and goodies move in the same way. The movement() function returns an int, either 0, 180, or -1. Returning -1 means the actor does not need to change their direction, while 0 and 180 indicate that an actor may need to turn to face that particular direction.

**Peach Class:**

I chose to define the functions doSomething(), bonk(), damage(), updateHit(int amt), hasStar(), hasShoot(), hasJump(), setStar(bool star), setShoot(bool shoot), and setJump(bool jump) in the Peach class as they are all related to Peach and the actions she can take. None of these functions were defined as virtual since no classes are derived from Peach.

* Peach(int x, int y, StudentWorld\* w);
  + Constructs a Peach object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Peach()
  + Virtual destructor for the Peach class. It is empty, but is included because there is inheritance and polymorphism
* virtual void doSomething();
* Since it is a pure virtual function in the Actor class and Peach is a derived class from Actor, I needed to implement the function for her to do anything during the game. The doSomething function is responsible for making sure Peach behaves correctly and executes the proper action during a single tick of the game.
* virtual void bonk();
  + I defined this function in the Peach class since while it is already defined and implemented in the Actor class, Peach has to take additional action when she is bonked by other actors and her action differs depending on what powers she may or may not have.
* virtual void damage();
  + I defined this function in the Peach class because while it is already defined and implemented in the Actor class, Peach has to take additional action when another actor attempts to damage her. The damage() function simply calls Peach’s bonk function since she reacts in the same way.
* void updateHit(int amt);
* I defined this function in the Peach class so that Peach’s health points, held by the private member variable m\_hit, can be updated accordingly if needed. This function simply sets her health points to the amount passed as a parameter.
* bool hasStar() const;
* bool hasShoot() const;
* bool hasJump() const;
  + I defined these functions in the Peach class which return whether Peach has star power, shoot power, and/or jump power respectively. These functions are const since they don’t make any modifications and simply return a boolean value.
* void setStar(bool star);
* void setShoot(bool shoot);
* void setJump(bool jump);
  + I defined these functions in the Peach class which set Peach’s private member variables determining her powers to star, shoot, and jump respectively.

**Stationary Class**

Stationary class encompasses the stationary objects in the game, which include blocks and pipes. Stationary is derived from the Actor class.

* Stationary(int ID, int x, int y, StudentWorld\* w);
  + Constructs a Stationary object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Stationary()
  + Virtual destructor for the Stationary class. It is empty, but is included because there is inheritance and polymorphism
* virtual void doSomething();
  + I defined this function in the Stationary class since Stationary is derived from Actor and the doSomething() function in Actor is pure virtual. Along with that, the two classes derived from Stationary, block and pipe, have the same behaviour when doSomething() is called. Since blocks and pipes don’t need to perform any action during a tick of the game, the function does nothing.

**Block Class**

Block class is derived from the class Stationary and has the function bonk()

* Block(int x, int y, StudentWorld\* w);
  + Constructs a Block object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Block()
  + Virtual destructor for the Block class. It is empty, but is included because there is inheritance and polymorphism
* virtual void bonk();
  + This function was defined and implemented in the Actor class, but the implementation in the Actor class does nothing when called. Thus, I implemented it in Block since blocks make a sound when they are bonked.

**GoodieBlock Class**

The GoodieBlock Class is derived from the class Block and is a base class for all blocks that hold goodies. This class was created since all goodie blocks have similar behaviour.

* GoodieBlock(int x, int y, StudentWorld\* w);
  + Constructs a GoodieBlock object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~GoodieBlock()
  + Virtual destructor for the GoodieBlock class. It is empty, but is included because there is inheritance and polymorphism
* virtual void bonk();
  + I defined this function in the GoodieBlock class which determines whether a block currently holds a goodie, and if the block does, it calls the addGoodie() function defined in each separate derived class
* virtual void addGoodie() = 0;
  + I defined this function in the GoodieBlock class since all GoodieBlocks have the ability to add a goodie to the game. However, I made it pure virtual since GoodieBlocks add different goodies to the game depending on what type of goodie they possess.

**StarBlock / MushBlock / FlowBlock Classes**

The StarBlock, MushBlock, and FlowBlock class are all derived from the GoodieBlock class and only have a constructor and destructor. They do have a private method addGoodie().

\* w);

* StarBlock(int x, int y, StudentWorld);
  + Constructs a StarBlock object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~StarBlock()
  + Virtual destructor for the StarBlock class. It is empty, but is included because there is inheritance and polymorphism
* MushBlock(int x, int y, StudentWorld);
  + Constructs a MushBlock object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~MushBlock()
  + Virtual destructor for the MushBlock class. It is empty, but is included because there is inheritance and polymorphism
* FlowBlock(int x, int y, StudentWorld);
  + Constructs a FlowBlock object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~FlowBlock()
  + Virtual destructor for the FlowBlock class. It is empty, but is included because there is inheritance and polymorphism

**Pipe Class**

The Pipe class is derived from the Stationary class and has no functions other than the constructor and destructor since all its behaviour is covered in either the Stationary or Actor class.

* Pipe(int x, int y, StudentWorld\* w);
  + Constructs a Pipe object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Pipe()
  + Virtual destructor for the Pipe class. It is empty, but is included because there is inheritance and polymorphism

**Character Class**

The Character class is derived from the Actor class and has the functions doSomething(), getSoundHit(), indivAct(), and getPoints(). It is the base class for the derived classes Finished and Goodies.

* Character(int id, int x, int y, int dir, int depth, double size, StudentWorld\* w, int points);
  + Constructs a Character object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Character()
  + Virtual destructor for the Character class. It is empty, but is included because there is inheritance and polymorphism
* virtual void doSomething();
  + I defined function in the Character class since all characters have similar behaviour when they do something and since doSomething() is defined as pure virtual in the Actor class, it needs to be implemented. Within the doSomething() function, the character’s indivAct() function is called since while the majority of the behaviour of characters is the same, each character has slightly different behaviour.
* int getSoundHit() = 0;
  + I definedfunction in the Character class since all characters make a sound when they are bonked/damaged by another Actor in the game. This function is defined as pure virtual since while all characters make a sound, their sounds are all different.
* virtual void indivAct() = 0;
  + I defined this function in the Character class since each character needs a function for the slightly different behaviour they have. This function is defined as pure virtual since while all characters need a function for their behaviour, each character performs different actions
* int getPoints() const;
  + I defined this function in the Character class since each character has a certain number of points that it awards the player when it overlaps with Peach. This function is neither virtual nor pure virtual since the function works the same for all characters and is const since it does not modify the character, but simply returns m\_points which is a private member variable of character.

**Finished Class**

The Finished class is derived from the Character class and has the public notify(). It serves as the base class for Mario and Flag.

* Finished(int id, int x, int y, StudentWorld\* w);
  + Constructs a Finished object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Finished()
  + Virtual destructor for the Finished class. It is empty, but is included because there is inheritance and polymorphism
* virtual void notify() = 0;
  + I defined this function in the Finished class since all Finished objects must notify the player when Peach overlaps with it. This function is defined as pure virtual since while all Finished objects need to notify the player, they notify the player of different things.

**Flag Class**

The Flag class is derived from the Finished class and has no public functions other than the constructor and destructor.

* Flag(int x, int y, StudentWorld\* w);
  + Constructs a Flag object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Flag()
  + Virtual destructor for the Flag class. It is empty, but is included because there is inheritance and polymorphism

**Mario Class**

The Mario class is derived from the Finished class and no public functions other than the constructor and destructor.

* Mario(int x, int y, StudentWorld\* w);
  + Constructs a Mario object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Mario()
  + Virtual destructor for the Mario class. It is empty, but is included because there is inheritance and polymorphism

**Goodies Class**

The Goodies Class is derived from Character and has the public setPow(bool pow). It serves as the base class for the classes Flower, Mushroom, and Star.

* Goodies(int id, int x, int y, StudentWorld\* w, int points);
  + Constructs a Goodie object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Goodie()
  + Virtual destructor for the Goodie class. It is empty, but is included because there is inheritance and polymorphism
* virtual void setPow();
  + I defined this function in the Goodie class since if a goodie overlaps with Peach, Peach must be notified that she now has that power. I defined this function as pure virtual since each goodie gives Peach a different power.

**Flower Class**

The Flower Class is derived from the Goodies class and has no public functions other than the constructor and destructor.

* Flower(int x, int y, StudentWorld\* w);
  + Constructs a Flower object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Flower()
  + Virtual destructor for the Flower class. It is empty, but is included because there is inheritance and polymorphism

**Mushroom Class**

The Mushroom Class is derived from the Goodies class and has no public functions other than the constructor and destructor.

* Mushroom(int x, int y, StudentWorld\* w);
  + Constructs a Mushroom object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Mushroom()
  + Virtual destructor for the Mushroom class. It is empty, but is included because there is inheritance and polymorphism

**Star Class**

The Star Class is derived from the Goodies class and has no public functions other than the constructor and destructor.

* Star(int x, int y, StudentWorld\* w);
  + Constructs a Star object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Star()
  + Virtual destructor for the Star class. It is empty, but is included because there is inheritance and polymorphism

**Enemies Class**

The Enemies class is derived from the Actor class and has the functions doSomething(), bonk(), and damage(). This class serves as the base class for the Koopa, Goomba, and Piranha classes.

* Enemies(int id, int x, int y, StudentWorld\* w);
  + Constructs an Enemies object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Enemies()
  + Virtual destructor for the Enemies class. It is empty, but is included because there is inheritance and polymorphism
* virtual void doSomething();
  + I defined this function in the Enemies class since some enemies have similar behaviour during a single tick of a game. This function is responsible for the movement and actions of an enemy during a single tick of the game.
* virtual void bonk();
  + I defined this function, which is responsible for an enemy’s behaviour when bonked by another actor, in the Enemies class since all Enemies have the same behaviour when bonked by another actor in the game.
* virtual void damage();
  + I defined this function, which is responsible for an enemy’s behaviour when another actor attempts to damage it, in the Enemies class since all Enemies have the same behviour when another actor attempts to damage it.

**Goomba Class**

The Goomba class is derived from the Enemies class and has no functions other than the constructor and destructor since all its behaviour is defined in either the Actor or Enemies class.

* Goomba(int x, int y, StudentWorld\* w);
  + Constructs a Goomba object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Goomba()
  + Virtual destructor for the Goomba class. It is empty, but is included because there is inheritance and polymorphism

**Koopa Class**

The Koopa class is derived from the Enemies class and has the function dead()

* Koopa(int x, int y, StudentWorld\* w);
  + Constructs a Koopa object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Koopa()
  + Virtual destructor for the Koopa class. It is empty, but is included because there is inheritance and polymorphism
* virtual void dead();
  + I defined function in the Koopa class since unlike other actors that simply set their alive status to false when they die, a Koopa must release a shell at its current position when it dies.

**Piranha Class**

The Piranha class is derived from the Enemies class and has the function doSomething()

* Piranha(int x, int y, StudentWorld\* w);
  + Constructs a Piranha object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Piranha()
  + Virtual destructor for the Piranha class. It is empty, but is included because there is inheritance and polymorphism
* virtual void doSomething();
  + I defined this function in the Piranha class since a Piranha’s behaviour during a single tick of the game is different from other enemy objects. The doSomething function checks for overlap with Peach or nearness to Peach and turns or fires a fireball accordingly.

**Projectile Class**

The Projectile class is derived from the Actor class and has the functions doSomething() and damageActor(). This class serves as the base class for PiranFire and PeachProj.

* Projectile(int id, int x, int y, int dir, StudentWorld\* w);
  + Constructs a Projectile object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Projectile()
  + Virtual destructor for the Projectile class. It is empty, but is included because there is inheritance and polymorphism
* virtual void doSomething();
  + function in the Projectile class since all Projectiles have the same behaviour during a single tick of a game. The function attempts to damage an actor if necessary and if unable to, attempts to move until it is blocked by another actor.
* virtual void damageActor() = 0;
  + I defined this function in the Projectile class since all projectiles have the ability to damage another actor, but it is pure virtual since different projectiles damage different actors.

**PiranFire Class**

The PiranFire class is derived from the Projectile class and has no public functions other than the constructor and destructor.

* PiranFire(int x, int y, int dir, StudentWorld\* w);
  + Constructs a PiranFire object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~PiranFire()
  + Virtual destructor for the PiranFire class. It is empty, but is included because there is inheritance and polymorphism

**PeachProj Class**

The PeachProj class is derived from the Projectile class and has has no public functions other than the constructor and destructor. This class also serves as the base class for the PeachFire and Shell classes.

* PeachProj(int id, int x, int y, int dir, StudentWorld\* w);
  + Constructs a PeachProj object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~PeachProj()
  + Virtual destructor for the PeachProj class. It is empty, but is included because there is inheritance and polymorphism

**PeachFire / Shell Class**

The PeachFire and Shell classes are derived from the PeachProj class and have no functions defined other than the constructor and destructor since their behaviour has been taken care of by base classes.

* PeachFire(int x, int y, int dir, StudentWorld\* w);
  + Constructs a PeachFire object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~PeachFire()
  + Virtual destructor for the PeachFire class. It is empty, but is included because there is inheritance and polymorphism
* Shell(int x, int y, int dir, StudentWorld\* w);
  + Constructs a Shell object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~Shell()
  + Virtual destructor for the Shell class. It is empty, but is included because there is inheritance and polymorphism

**StudentWorld Class**

The StudentWorld class is derived from GameWorld and consists of 11 functions other than init(), move(), and cleanup(). None of the functions in StudentWorld are created as virtual or pure virtual since there are no classes derived from StudentWorld. However, init(), move(), and cleanup() are pure virtual in the GameWorld class and have their own implementations in StudentWorld.

* StudentWorld(std::string assetPath);
  + Constructs a StudentWorld object and is neither virtual nor pure virtual since it is a constructor.
* virtual void ~StudentWorld()
  + A destructor for StudentWorld which deletes all actors in the game by calling StudentWorld’s cleanup() function. It is virtual since there is inheritance and polymorphism.
* void addActor();
  + I defined this function which takes in a pointer to an Actor and adds it to the vector containing all actors in the game.
* void setLevComp(bool level); / void setGameComp(bool level)
  + I defined this two functions which have similar uses. They both take in a bool and set the private variables m\_levelComp and m\_game Comp to that bool value respectively. I chose to create this function so that if one of the actors determines that a level/game should be finished, it can notify the StudentWorld class.
* Peach\* getPeach()
  + I defined this function that returns a pointer to Peach. I created this function so that actors can have easier access to Peach without needing to call many accessor or mutator methods.
* bool overlaps(int x, int y, int action);
  + I defined this function that takes in a set of coordinates and an int determining whether action should be taken if there is overlap between two actors. This function calls the helper function checkBounds. It determines if there is overlap and if necessary, bonks/damages the actor that blocks the path. It then returns whether action has been taken or successfully executed.
* bool peachOverlap(Actor\* a)
  + I defined this function that has essentially the same functional purpose as the overlaps function except that it takes in a pointer to an actor and compares whether that particular actor overlaps with Peach. This function also calls the helper method checkBounds.
* bool peachBonk(Actor\* a)
  + I defined this function that takes in a pointer to an actor and bonks Peach if there is overlap between the actor and Peach. This function calls the peachOverlap function to determine if there is overlap, and if so, calls Peach’s bonk function.
* virtual int init()
  + This function was declared as pure virtual in the GameWorld class and thus, I implemented it in StudentWorld. The purpose of this function was to load the level file and add the appropriate actors the appropriate positions in the game. In the init function, I called the loadLevel function to acquire the proper level text string. The function then looped through the entire level file and input the proper actors in their respective places.
* virtual int move();
  + This function was declared as pure virtual in the GameWorld class and thus, I implemented it in the StudentWorld class. The move function calls Peach’s doSomething function and then loops through all the actors that were present in the game at the beginning of the game tick and calls their doSomething function if they are alive. If an actor is added during a tick, their doSomething function is not called. This function also determines when a player dies and when the game/level ends.
* virtual void cleanup();
  + This function was declared as pure virtual in the GameWorld class and thus, I implemented it in the StudentWorld class. This function loops through the vector m\_actors that contains all the actors of the game using an iterator. It deletes all references to any actors in the game.

2. After playing the game multiple times, to my knowledge, there are no bugs with the behaviour of the different actors.

3.

Design Decisions

* Function in StudentWorld returning a pointer to Peach
  + I originally had multiple accessor and mutator methods in StudentWorld to make changes to Peach’s powers and abilities. I created these methods since I thought we were not allowed to return a pointer to Peach per OOP tip #7. However, after verifying with Professor Smallberg that returning a pointer to Peach to allow other actors to access her within Actors.cpp was allowed, I created a method that returns a pointer to Peach so that actors have easier access to her. This also allowed me to get rid of the many accessor and mutator methods that were in the StudentWorld class.
* checkBounds Function
  + I created a helper method checkBounds that takes two sets of coordinates in as parameters. This method checks and returns whether those two coordinates overlap. I chose to create this helper function since there are multiple instances in which the bounds of two actors needs to be compared and it wouldn’t make sense to duplicate the same code multiple times. Thus, I created this method that is simply called by other functions when overlap needs to be determined.
* Separate PeachBonk() and PeachOverlap() functions
  + In StudentWorld, I already have an overlap function which determines whether two actors overlap and bonks/damages the actor being overlapped accordingly. However, I found that it would be easier and more organised to have a separate function that determined overlap with Peach and one that bonks Peach when she is overlapped. I was able to do this because of the previous design decision described (the checkBounds function). Since overlapping and bonking Peach is unique from other actors, I found it helpful to create separate functions for her.
* Not calling doSomething of newly added characters
  + In StudentWorld, I originally used an iterator to loop through the actors and call their doSomething functions. However, I ran into the issue of having a bad access error when adding Shells into the game in the middle of a tick. To resolve this error, I created a variable to hold the size of the vector at the beginning of the tick. I then looped through those actors and called their doSomething function. Since the spec outlined that we could choose whether an actor should call doSomething during the tick in which they’re added, I chose to not call doSomething for newly added actors.
* Creating global constants to keep track of what should happen if two actors overlap
  + Since the overlap function will be called to determine overlap and there may be some instances where an actor should be bonked or damaged, I chose to create three global constants. These global constants were defined in Actor.h. ACTION\_BONK is passed to the overlap function when an actor should be bonked when there is overlap. ACTION\_DAMAGE is passed to the overlap function when an actor should be damaged when there is overlap. NO\_ACTION is passed to the overlap function when no action should be taken and the overlap function is simply being called to determine whether there’s overlap or not
* The pure virtual functions defined in base classes that were later implemented in derived classes were made private in the derived classes if they were not called by any other classes. For example, in the derived classes of Projectile, the function damageActor() was made private since no other class other than Projectile calls that function.

Assumptions

* One thing I was slightly confused about in the spec was Peach checking whether she had the ability to fall. In the spec, it said to check 0-3 pixels below her, and if there was nothing blocking her at those pixels, she should move 4 pixels down. I didn’t quite understand why we only checked up to and including 3 pixels below her when she moves 4 pixels down, but since it did not lead to any issues implementing it this way, I followed the spec and checked 0-3 pixels below Peach.
* There was a discrepancy between the spec and the sample game as to whether a sound should be played when a pipe is bonked. I chose to follow the spec and thus, no sound is made when a pipe is bonked.
* The specification was unclear as to whether the powerups that Peach attains should be displayed in a certain order, such as the order in which she attains them. However, after playing the sample game, it seemed that the order was unrelated to the order in which she attained them, so I chose to follow the format of the sample game, where the powers will be displayed in the order of StarPower, ShootPower, and JumpPower.