Computing IPM Analyses

# Ashley Solano Hernandez 9C

Class:

Description:

|  |  |
| --- | --- |
| public class PlayGround extends World  {    /\*\*  \* Constructor for objects of class PlayGround.  \*  \*/  public PlayGround()  {  super(1000,600,1);  start();  }  private void start()  {  addObject(new Game(), 250, 25);  }  } | This is the constructor for the class PlayGround.  The method super sets the world size to a rectangle of 1000 by 600 squares, each being one pixel in length.  There is also another method within the constructor, the method is called start().  Method called start(), only objects which are of class PlayGround can use it due to it being private.  The method start() adds an object of class Game at the coordinates of 250, 25 |

|  |  |
| --- | --- |
| public abstract class AbstractHero extends Actor  {  protected int health;  protected int speed;  protected Weapon currentWeapon;  protected GreenfootSound heroSpawn = new GreenfootSound("Hero-ok.wav");  protected GreenfootSound heroDeath = new GreenfootSound("Hero-death.wav");  public void act() {  move();  shoot();  collide();  }    public int getHealth() {  return this.health;  }    public void move()  {  if(Greenfoot.isKeyDown("up"))  {  setLocation(getX(),getY()-speed);  }  if(Greenfoot.isKeyDown("down"))  {  setLocation(getX(),getY()+speed);  }  if(Greenfoot.isKeyDown("right"))  {  move(speed);  }  if(Greenfoot.isKeyDown("left"))  {  move(-speed);  }  }    public void shoot()  {  if(Greenfoot.isKeyDown("space"))  {  getWorld().addObject((Actor)this.currentWeapon, 0, 0);  this.currentWeapon.shoot(getX(),getY());  }  }    public void collide()  {  Actor actor;  actor = getOneObjectAtOffset(10,10, AbstractEnemies.class);  if (actor != null)  {  AbstractEnemies enemy = (AbstractEnemies) actor;  this.health = this.health - enemy.getDamage();  getWorld().removeObject(enemy);  die();  }  }    public void die() {  if (health < 0) {  java.util.List gameList = getWorld().getObjects(Game.class);  Game game = (Game)gameList.get(0);  heroDeath.setVolume(100);  heroDeath.play();  game.stop();  getWorld().removeObject(this);  }    }  } | This is an abstract class called AbstractHero and it is abstract as it can’t be created directly but can be extended.  The variable called health is of type integer. Only objects of class AbstractHero and its subclasses can use it due to it being protected.  The variable called speed is of type integer. Only objects of class AbstractHero and its subclasses can use it due to it being protected.  The variable called currentWeapon is of type Weapon. Only objects of class AbstractHero and its subclasses can use it due to it being protected.  The variable called heroSpawn is of type GreenfootSound and has a value of "Hero-ok.wav". Only objects of class AbstractHero and its subclasses can use it due to it being protected.  The variable called heroDeath is of type GreenfootSound and has a value of "Hero-death.wav". Only objects of class AbstractHero and its subclasses can use it due to it being protected.  The “act” method is called by Greenfoot every loop. Inside of the “act” method are the move(), shoot() and collide() methods.  The getter for the health property.  The move() method is used to move the hero, it can be used by all classes as it is a public method.  Inside the move() method, there lines of code which changes the hero’s X or Y position.  The shoot() method is used to shoot bullets, it can be used by all classes as it is a public method.  Inside the shoot() method adds an object of class Weapon to the world and invokes its shoot every time the space key is pressed.  The collide() method is used to detect if the hero comes into contact with an enemy, it can be used by all classes as it is a public method.  Inside the collide() method, a variable of type actor is created and is used to detect if an actor of type AbstractEnemies is within a 10,10 radius of the hero.  If the hero detects an enemy within a radius of 10,10 from the hero, then a variable called enemy of type AbstractEnemies is created and casts the actor to AbstractEnemies.  The health of the hero is then reduced by the variable called damage which belongs to the class AbstractEnemies.  The actor that touched the hero is then removed from the world.  The die() method is then processed.  The die() method is used to check if the hero should be removed from the world.  If the health of the hero is lower than 0, then the sound called heroDeath will play, the game stop will be invoked and the hero will be removed from the world. |

|  |  |
| --- | --- |
| public class Zoom extends AbstractHero  {  public Zoom (){  super ();  this.health = 15;  this.speed = 10;  this.currentWeapon = new MachineGun();  heroSpawn.setVolume(100);  heroSpawn.play();  }  } | This is a class called Zoom and inherits all of the protected methods of AbstractHero.  Inside of the Zoom constructor, his health, speed and weapon are defined, the heroSpawn sound is also played. |

|  |  |
| --- | --- |
| public class AbstractBullets extends Actor implements Ammo  {  protected int speed;  protected int damage;  protected boolean active = true;    public void act() {  move();  exitWorld();  collision();  }    public int getDamage(){  return this.damage;  }    public AbstractBullets(int pSpeed) {  speed = pSpeed;  }    public void move() {  move(speed);  }    public void exitWorld() {    if (this.active && getX() >= 995) {  getWorld().removeObject(this);  this.active = false;  }  }  public void collision(){  if(this.active ) {  getWorld().removeObject(this);  this.active = false;  }  }  } | This class is called AbstractBullets and extends to the Actor class and implements ammo.  The Boolean variable, active and the integer variables of speed and damage are declared, all of them are protected so subclasses of AbstractBullets can utilise them.  The act() method contains the move(), exitWorld() and collision() methods.  The damage getter gets how much damage a bullet does.  The speed getter gets how fast a bullet moves.  The move() method makes a bullet move to the right at a rate of its speed.  The exitWorld() method makes the bullet disappear if it is active and if it goes beyond the X coordinate of 995 and declares that it is no longer active.  The collision() method makes the bullet disappear if it is active. |

|  |  |
| --- | --- |
| public class MagicBullet extends AbstractBullets  {    public void act() {  move();  exitWorld();  }    public MagicBullet() {  super (10);  damage = 2;  }  }  }  }    public void die() {  java.util.List gameList = getWorld().getObjects(Game.class);  Game game = (Game)gameList.get(0);  if (this.active && this.health < 1 ) {  this.active = false;  World world;  world = getWorld();  world.removeObject(this);  game.increaseScore(this.points);  }    }  } | The class MagicBullet extends to AbstractBullets.  The act() method contains the move() and exitWorld() methods.  In the MagicBullet constructor, the speed integer parameter is defined as 10 and its damage as 2.  The die() method checks if the bullet’s life is under 1 and if it is active, if it is, then it will make active false and will remove be removed from the world and it will also increase the score variable belonging to the class called Game. |

|  |  |
| --- | --- |
| public class AbstractEnemies extends Actor  {  protected int health;  protected int speed;  protected int damage;  protected boolean active;  protected int points;  public AbstractEnemies() {  this.active = true;  }  /\*\*  \* Act - do whatever the Enemy wants to do. This method is called whenever  \* the 'Act' or 'Run' button gets pressed in the environment.  \*/  public void act()  {  move();  collide();  exitWorld();  }    public int getDamage(){  return this.damage;  }    private void move()  {  move(-speed);    }    private void exitWorld()  {  if ( this.active && getX()<=1) {  this.active = false;  getWorld().removeObject(this);    }  }    public void collide()  {  Actor actor;  actor = getOneObjectAtOffset(4,4, AbstractBullets.class);  if (actor != null)  {  AbstractBullets bullets = (AbstractBullets) actor;  this.health = this.health - bullets.getDamage();  die();  bullets.collision(); | This is the AbstractEnemies which extends to Actor.  It has health, speed, damage and points as integers and active as a Boolean. They are protected so all its subclasses can use these variables.  It starts off with having active as true.  In its act() method, it has the move(), collide() and exitWorld() methods.  In its damage getter, it gets how much damage an enemy deals.  In its move() method, it moves to the left at the speed of the variable “speed.”  Its exitWorld() method destroys the enemy if it touches the world boundary and if it is active, it also makes active turn false.  The collide() method checks if a bullet is within a 4 by 4 radius of the enemy, if it is, then gets the amount of damage that the bullet deals and takes it away from his health.  It will carry out the die() method if its health is under 0. |

|  |  |
| --- | --- |
| public class HomingEnemy extends AbstractEnemies  {  public HomingEnemy (){  health = 3;  speed = 5;  damage = 5;  points = 5;  }  } | This is the class HomingEnemy which extends to AbstractEnemies  It has set health=3,speed=5, damage=5, points=5. |

|  |  |
| --- | --- |
| public class Enemy extends AbstractEnemies  {  public Enemy (){  health = 1;  speed = 4;  damage = 2;  points = 2;  }  } | This is the Enemy class which extends to AbstractEnemies  In its constructor, it has set health = 1, speed = 4, damage = 2, points = 2. |

|  |  |
| --- | --- |
| public class Game extends Actor  {  private AbstractHero currentHero;  private boolean gameInProgress = false;  private double lastSpawnTime = System.currentTimeMillis();  private double spawnRate = 450;//in milliseconds  private int score;  private GreenfootSound music = new GreenfootSound("Bizet.wav");  private GreenfootImage img;  private Random randomGenerator;  private Random randomYGenerator;    public void updateScore() {  if (this.gameInProgress) {  this.img = new GreenfootImage (200, 20);  this.img.drawString ("Life: " + this.currentHero.getHealth() + " Score: " + this.score ,2,20);  setImage(this.img);  }  }  public void increaseScore(int points) {  this.score = this.score + points;  }    /\*\*  \* Act - do whatever the Game wants to do. This method is called whenever  \* the 'Act' or 'Run' button gets pressed in the environment.  \*/  public void act()  {  start();  updateScore();  spawnEnemies();  }    private void start() {  if("s".equals(Greenfoot.getKey()) && !this.gameInProgress)  {  spawnHero();  spawnEnemies();  this.gameInProgress = true;  this.randomGenerator = new Random();  this.randomYGenerator = new Random();  music.setVolume(80);  music.play();  this.score = 0;  getWorld().setBackground("background0172.jpg");  }  }    public void stop() {  this.gameInProgress = false;  this.music.stop();  getWorld().setBackground("desert.png");  this.img = new GreenfootImage (1000,600);    this.img.drawString ("GAME OVER! ",500,300);  this.img.drawString ("Try again! ",500,320);  setImage(this.img);    }    private void spawnHero()  {  this.currentHero = new Zoom();  getWorld().addObject(this.currentHero, 100, 250);  }    private void spawnEnemies() {  double currentSpawnTime = System.currentTimeMillis();  double timeElapsed = currentSpawnTime - this.lastSpawnTime;    if ((this.gameInProgress) && (timeElapsed >= this.spawnRate)) {    int randomEnemyType = 1 + this.randomGenerator.nextInt(100 - 1 + 1);  int randomY = 10 + this.randomGenerator.nextInt(590 - 10 + 1);    if (randomEnemyType > 10) {  getWorld().addObject(new Enemy(), 1000, randomY);  this.lastSpawnTime = System.currentTimeMillis();  }  else {  getWorld().addObject(new HomingEnemy(), 1000, randomY);  this.lastSpawnTime = System.currentTimeMillis();  }  }  }  } | This is a class called Game which extends to Actor.  It has variables of type: AbstractHero type called currentHero, boolean type called gameInProgress, double called lastSpawnTime and is assigned how long the game has been running for, another double called spawnRate which has a value of 450 milliseconds, integer called score.  There are also variables of type: GreenfootSound called music which has a value of the Greenfoot sound "Bizet.wav", GreenfootImage called img and two Random called randomGenerator and randomYGenerator which generate random numbers.  All the variables of Game are private, so only objects of type Game can use them.  The updateScore() method creates a new image and sets a new image for the actor Game with the size of 200 by 20 with the text “Life:” the current hero’s health and “ Score:” and the current score.  The increaseScore(int points) has a parameter which is an integer variable, it is then added to the current score to get the new score.  Inside the act() method, there is the start() updateScore() and spawnEnemies() methods.  The start() method invokes the spawnHero() and the spawnEnemies() methods, makes gameInProgress true, starts the two number generators making random values. It also sets the music volume to 80 and then plays the music, it sets the value of score to 0 and changes the world background.  All of this happens when the s button is pressed and if the gameInProgress variable is false. Anyone can use this method as it is public.  The stop() method sets gameInProgress to false, it also makes the music stop, changes the world background and makes a new image which says GAME OVER!!! And Try again! On another line and sets I as the image for Game.  The spawnHero() method makes the world add Zoom to the world.  The spawnEnemies() method controls the frequency with which enemies spawn.  Inside the method, a new variable is created with the value of the current game time, another variable called timeElapsed compares the last time since an enemy was spawned to the current game time, if the game is in progress and there is a difference = to or > than spawnRate, then a new enemy is created.  To create an enemy, the game class makes a random number between 1 and 100 which is assigned to randomEnemyType and another random number between 10 and 590 which is assigned to randomY.  If randomEnemyType is greater than 10, then it will create an Enemy at 1000, randomY, the last time and enemy was spawned in is updated.  If randomEnemyType is not greater than 10, then It will create a HomingEnemy at 1000, randomY, the last time and enemy was spawned in is updated. |

|  |  |
| --- | --- |
| public abstract class AbstractWeapon extends Actor implements Weapon  {    private double lastShotTime = System.currentTimeMillis();  private double firingRate;//in milliseconds    private Ammo ammo;  private int ammoSpeed;    public AbstractWeapon(double pFiringRate, Ammo pAmmo, int pSpeed) {  this.firingRate = pFiringRate;  this.ammo = pAmmo;  this.ammoSpeed = pSpeed;  }    public void shoot(int pX, int pY) {    double currentShotTime = System.currentTimeMillis();  double timeElapsed = currentShotTime - this.lastShotTime;    if (timeElapsed >= this.firingRate) {  try {  String className = ammo.getClass().getSimpleName();  Actor ammoActor = (Actor) Class.forName(className).getConstructors()[0].newInstance();;    getWorld().addObject(ammoActor, pX, pY);  this.lastShotTime = System.currentTimeMillis();  GreenfootSound music = new GreenfootSound("Fireball.wav");  music.setVolume(90);  music.play();  } catch (Exception e) {  }  }  getImage().setTransparency(0);  }  } | The class AbstractWeapon is an abstract class, so it can’t be created directly, extends to Actor and implements Weapon.  It has a double variable to check the last time the weapon shot a bullet, the variable is called lastShotTime.  It also declares: a variable of type double called firingRate, a variable of type Ammo called ammo, a variable of type integer called ammoSpeed.  All of the variables are private, so only actors of class AbstractWeapon can use them.  The constructor for AbstractWeapon has 3 parameters which are a double firing rate, a specific ammo type and a speed integer to which they are assigned to permanent variables.  The shoot() method has a parameter of an X and Y coordinates and is used to shoot bullets.  A new variable is created called currentShotTime which is assigned to the current game time, another variable called timeElapsed finds the difference between the current time at which it tried to shoot and the last time it shot a bullet.  If timeElapsed is greater than the firing rate, then it will create an actor of a specified bullet type at the location of the hero.  The lastShotTime is updated and a sound called Fireball.wav is played.  I surrounded this method in a try/catch to capture any exceptions thrown by this method. |

|  |  |
| --- | --- |
| public class MachineGun extends AbstractWeapon  {    /\*\*  \* Act - do whatever the MachineGun wants to do. This method is called whenever  \* the 'Act' or 'Run' button gets pressed in the environment.  \*/  public void act()  {  }  //AbstractWeapon(double pFiringRate, Ammo pAmmo, int pSpeed) {  public MachineGun() {  super (350, new MagicBullet(), 10);  }  } | This is a class called MachineGun which extends to AbstractWeapon  Its act() mthod is empty but its constructor sets the weapon’s firing rate, ammo type and bullet speed. |

# Evaluation of My Progress

I learned object oriented programming concepts with much more confidence such as:

* class
* constructors
* getters and setters
* instantiation
* initialisation
* inheritance
* objects

I learned the main aspects of Java’s syntax.

I learned how to use the version control software called git and its main concept of version control which were:

* creating a repository in github
* cloning a repository from github
* using command prompt
* commiting changes
* pushing changes to a remote repository

# Challenges Encountered

I had to learn a new methodology and thought process to use an Object-Oriented Program which took some time as I was accustomed to only using procedural programming.

I also found Greenfoot’s learning curve quite hard at the beginning.

The Oracle website wouldn’t load the Greenfoot courses, so I had to learn from other websites.

# What I would do with additional investment

If I were to receive additional funding for the programming of this game, then I would hire:

* Writers to create an interesting story behind the game
* Additional programmers so that they could help me program the game and allow it to be developed across multiple platforms
* Graphics designers so that the game had a compelling art style
* I would also commission original music scores for the game

# Ashley Solano Hernandez 9C