CSE-271: Object-Oriented Programming

Homework #4: Ball destruction

Phase #1: Wed Oct 19 2022 before 11:59 PM

Phase #2: Wed Oct 26 2022 before 11:59 PM

Email-based help Cutoff: 5:00 PM on Tue before deadline

Delayed (by no more than 24-hours) submissions earn only 80% credit

Maximum Points: 40

Key objectives of this project are:

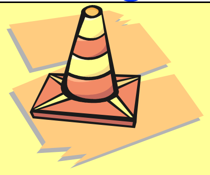
- Gain experience with developing and working with a class hierarchy
- Review concepts of creating custom classes
- Work with Graphical User Interface (GUI)
- Review and adhere to CSE department's Style guide
- Use Javadoc to document methods and their return values

Submission Instructions

Each phase of this project must be turned-in electronically via Canvas CODE plug-in. Ensure your program compiles successfully, without any warnings or style errors. Ensure you have documented the methods. Ensure you have tested operations of your classes.

General Note: Upload each file associated with homework (or lab exercises) individually to Canvas. Do not upload archive file formats such as zip/tar/gz/7zip/rar etc.

Grading Rubric:



The source code submitted for this homework **must pass necessary test(s) in order to qualify for earning any score at all**. Programs that do not meet basic requirements or just skeleton code will be assigned zero score! Programs that do not compile, **have even 1 method longer than 25 lines**, or just some skeleton code will be assigned zero score.

- **NOTE:** Violating CSE programming style guidelines is an error! Your program should not have any style errors.
- **Delayed submission: Only 80% points:** Submission delayed by no more than 24-hours will be accepted for a partial credit of maximum 80% of the points.
- **Conciseness, Formatting & Documentation: 4 points** – Reserved for concise solution, good Javadoc, and formatting in each phase. These points may be the hardest to earn!
- **Phase #1**: 4 points per class – 4×4=**16 points**.
- **Phase #2**: **16 points**.
- **Extra credit**: Maximum of 7-points. Must be documented with screenshot(s).

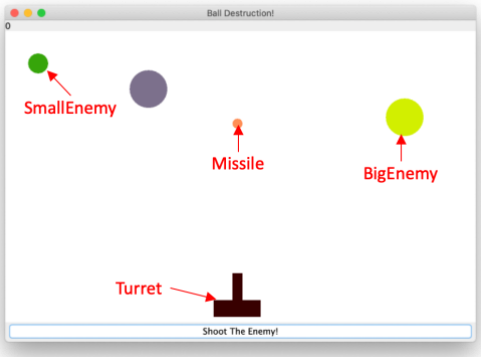
Project Overview:

In this project, you will be creating a basic video game using the knowledge of Object-oriented Programming (OOP) and Java Swing components you have learned so far. You will be given starter code to assist you in developing the game. Your goal is to develop classes and methods to make the game functional. This video game was envisioned by Dr. Garrett Goodman.

Extra credit: There will be extra credit assigned based on additional feature(s) added to the game, to create a unique game. The uniqueness can come from special rules, graphics, etc. but it must be substantive to earn full credit. If you are unsure, discuss your idea with your instructor. You must briefly describe the added functionality along with one-or-more screenshot(s). You can earn 1-to-7 additional points of extra credit. You may use the extra credit to improve your grads as you see fit – *e.g.*, add points to an exam or homework project (as long as the total does not exceed 100%).

Overview of the game

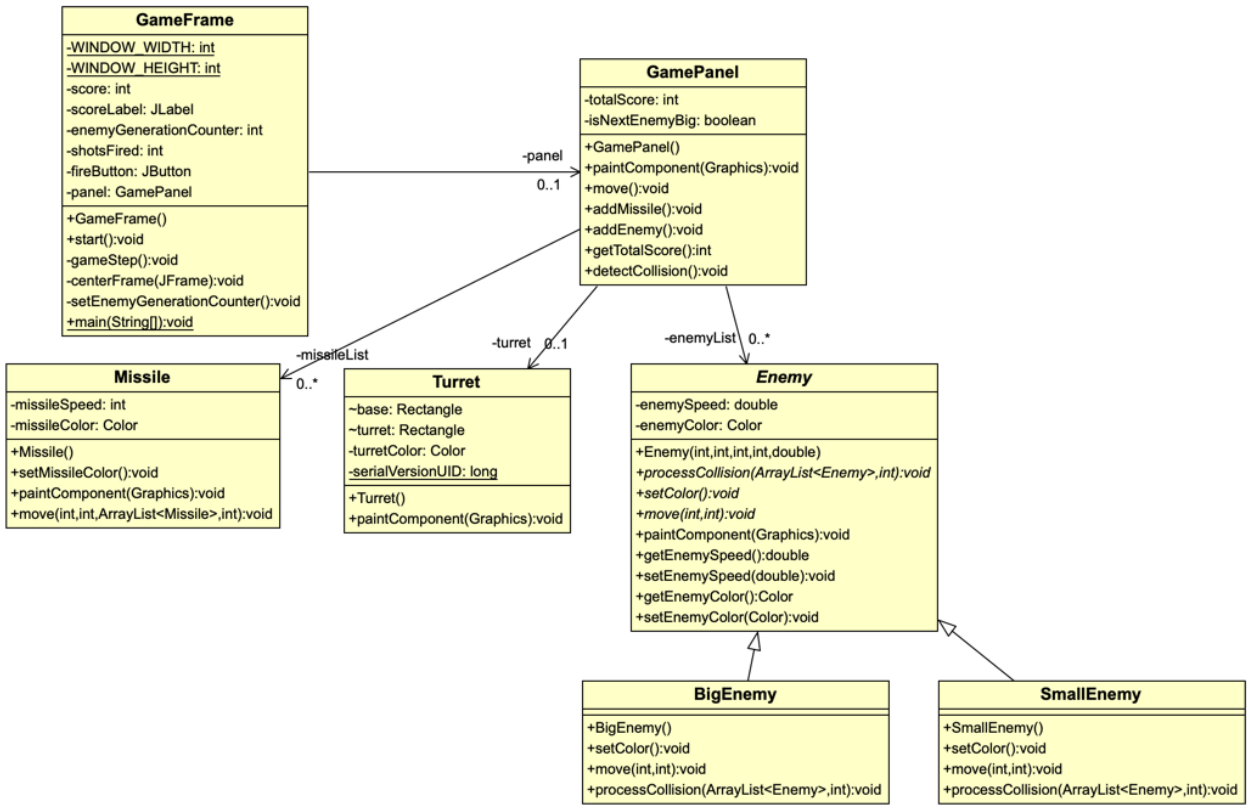
The objective of this project is to develop strength in OOP by developing a video game. A minimal version of the game is shown in the adjacent screenshot. The objective of the game is to shoot enemies to score points using a fixed number of missiles. The minimal ruleset of the game is simple. You press the button to shoot a Missile. If you hit a `BigEnemy`, then you get 100 points. If you hit a `SmallEnemy`, then you get 150 points. You can shoot a total of 10 Missiles and then the game ends. If your score is 800 or above, you win, else you lose. Note that these are suggested rules and you may create your own ruleset



Overview of the classes constituting the game

The UML diagram for the required classes in the game is shown on the next page. Note that there are two types of enemies, `SmallEnemy` and `BigEnemy`, both derived from an abstract `Enemy` class that contains common functionality. There is a separate `Missile` to hit the enemies. The `GameFrame` class extends `JFrame` and is the top-level class for this game. The `GameFrame` contains the following components of the game:

1. A `JLabel` keeps track of the score at the top of the `JFrame`.
2. A `JButton` at the bottom of the `JFrame` allows us to shoot a `Missile`.
3. A `GamePanel` class (is at the center of the `JFrame`) that encapsulates enemies (both big and small), missiles, and a `Turret`. The `Turret` is painted at the bottom center, more for display purposes to show the player where the missiles start when fired.



Phase #1: Develop core classes in the game

In order develop this game, the first step is to implement the core classes constituting the game, namely: `Enemy`, `BigEnemy`, `SmallEnemy`, and `Missile` as documented below.

`Enemy` class

This class is *abstract* and extends `JComponent`. The `JComponent` essentially provides features to the bounds of the object (see methods in `JComponent` such as: `setBounds`, `getX`, `getWidth`, etc.) and draw it. The `Enemy` class encapsulates the information common to both of its child classes (`BigEnemy` and `SmallEnemy`) and provides the following methods (see UML for details):

1. `Enemy(int x, int y, int height, int width, double enemySpeed)` – A constructor that initializes the bounds and speed.
2. `void processCollision(ArrayList<Enemy> list, int enemy)` – An *abstract* method (*i.e.*, no body) which determines what occurs when a `Missile` hits an Enemy. This method is overridden in derived classes.
3. `setColor()` – An *abstract* method which generates and sets the color of the `Enemy`. This method is overridden in derived classes.
4. `move(int frameWidth, int frameHeight)` – An *abstract* method which computes and updates the next position of the `Enemy`. The bounds of the enemy must always remain be within the specified `frameWidth` and `frameHeight`.
5. `paintComponent(Graphics g)` – Draws a filled circle using the `Enemy`'s color and its bounds (see `getBounds`).
6. Getters and setters as shown in the UML.

`BigEnemy` class

This class extends the `Enemy` and implements the abstract methods so that `BigEnemy` can be instantiated during game play. See UML diagram for details.

1. `BigEnemy(int panelWidth, int panelHeight)` – Creates an enemy of size 56×56 with randomly set `x` and `y` coordinates. However, the bounds must not exceed `panelWidth` and `panelHeight`. The speed is initialized to 4. The color is initialized by calling the `setColor` method.
2. `setColor()` – Randomly generate a color.
3. `move(int frameWidth, int frameHeight)` – Changes only the x-coordinate by adding `enemySpeed`. However, prior to changing the x-coordinate, if the addition will cause the x-coordinate to be higher than `frameWidth` or less than zero, then reverse the `enemySpeed` (i.e., multiply by `-1`). Ensure you update the bounds.
4. `processCollision(ArrayList<Enemy> list, int bigEnemy)` – This method is called when a `Missile` hits an enemy. Decrease current size by 20. If the `width` or `height` is less than zero, remove the enemy at index position `bigEnemy` from the `list`.

`SmallEnemy` class

This class extends the `Enemy` and implements the abstract methods so that `SmallEnemy` can be instantiated during game play. See UML diagram for details.

1. `SmallEnemy(int panelWidth, int panelHeight)` – Creates an enemy of size 30×30 with randomly set `x` and `y` coordinates. However, the bounds must not exceed `panelWidth` and `panelHeight`. The speed is initialized to 6. The color is initialized by calling the `setColor` method.
2. `setColor()` – Randomly generate a color.
3. `move(int frameWidth, int frameHeight)` – Changes only the x-coordinate by adding `enemySpeed`. However, prior to changing the x-coordinate, if the addition will cause the x-coordinate to be higher than `frameWidth` or less than zero, then reverse the `enemySpeed` (i.e., multiply by `-1`). Ensure you update the bounds. Finally, increase `enemySpeed` (both in positive and negative direction) by `0.05` (yes, each time the `move` method is called).
4. `processCollision(ArrayList<Enemy> list, int smallEnemy)` – This method is called when a `Missile` hits an enemy. Decrease current size by 10. If the `width` or `height` is less than zero, remove the enemy at index position `smallEnemy` from the `list`.

Missile class

This class extends `JComponent` (see methods in `JComponent` such as: `setBounds`, `getX`, `getWidth`, etc.) and encapsulates the information about a `Missile`. The methods in this class are to be implemented as documented below. See UML class diagram for additional details.

1. **`Missile(int x, int y)`** – The constructor to initialize a 15×15 missile at given `x` and `y` coordinates. The `missileSpeed` is set to 5. The color is randomly chosen. Note that the `x` and `y` coordinates correspond to the tip of the turret.

2. **`paintComponent(Graphics g)`** – Draws a filled circle using the `Missile`'s color and its bounds (see `getBounds`).
3. **`move(int panelWidth, int panelHeight, ArrayList<Missile> list, int missile)`** – Determine if the missile is off the panel and remove it from the `ArrayList` if it is. Also, determine the next position by changing just the `y`-coordinate by subtracting `missileSpeed`.

Phase #1: Submission via Canvas

For Phase #1 you don't need to have the full game developed. You only need to have the above classes developed. Use the supplied `Phase1Test.java` JUnit class to test your classes.

Once you have tested your classes, upload the following Java source files to Canvas: `Enemy.java`, `BigEnemy.java`, `SmallEnemy.java`, and `Missile.java` to Canvas via the CODE plug-in.

Phase #2: Complete the game & add extra credit features

In this phase you will be using the classes you developed in Phase #1 to complete the game. Optionally, you may add (and document) extra features. In this phase, you may modify the classes any way you see fit. It is all up to you.

Turret class

In the basic version of the game this component is just primarily for display purposes. This class inherits `JComponent` and contains the following information:

1. **Instance variables:**
   a. `Rectangle base` – The rectangle that represents the base of the `Turret`. You may decide on the bounds to make it look good.
   b. `Rectangle turret` – The rectangle that represents the turret of the `Turret`. You may decide on the bounds to make it look good.
   c. `Color turrentColor` – The color of the `Turret`. You may decide on the color.
2. **Methods:**
   a. `Turret()` – The default (or no-argument) constructor to initialize the instance variables.
   b. `paintComponent(Graphics g)` – The `paintComponent()` method that paints the `Turret` base and `Turret` barrel.

The GamePanel class

This class inherits `JPanel` and contains all the GUI elements associated with the game. See the UML class diagram for details.

1. **Instance variables:**
   a. `int totalScore` – The score to be updated by the game.
   b. `boolean isNextEnemyBig` – A Boolean to determine which enemy to generate next, either a `BigEnemy` or `SmallEnemy`.

c. `Turret turret` – A single `Turret` object to use throughout the game.
d. `ArrayList<Enemy> enemyList` – A polymorphic `ArrayList` to hold both `BigEnemy` and `SmallEnemy` objects that are currently visible in the game.
e. `ArrayList<Missile> missileList` – An `ArrayList` to hold `Missile` objects that are currently visible in the game.

2. **Methods:**
   a. `GamePanel()` – Constructor to initialize the instance properties. It also adds one new `BigEnemy` and a `SmallEnemy` to the `enemy ArrayList` to start the game.
   b. `paintComponent(Graphics g)` – For every `Enemy`, every `Missile`, and the Turret using their respective `paintComponent` methods.
   c. **move** – This method is called to animate the game. Move each `Enemy` and every `Missile` using their respective `move` method.
   d. `addMissile()` – Adds a new `Missile` to the game. Remember to add it to the `missileList`.
   e. `addEnemy()` – Adds a new `BigEnemy` or `SmallEnemy` to the `enemyList` depending on the `isNextEnemyBig`. The value of `isNextEnemyBig` is toggled.
   f. `getTotalScore()` – Returns the `totalScore` instance property.
   g. `detectCollision()` – This method is already implemented for you. It detects if a `Missile` hits an `Enemy` and updates the score. The `Missile` that hit the enemy is removed and the `Enemy` is affected by calling the `processCollission`() method in the corresponding `Enemy` object.

The `GameFrame` class

This class inherits `JFrame` and contains all the information necessary to display the GUI and to have the game operate. See the UML class diagram for additional details.

a. **Instance Properties:**
   i. `WINDOW_WIDTH` – Suggested width of the `JFrame`.
   ii. `WINDOW_HEIGHT` – Suggested height of the `JFrame`.
   iii. `int missilesFired` – The number of Missiles fired.
   iv. `JLabel scoreLabel` – The `JLabel` to display the score.
   v. `JButton fireButton` – The `JButton` to fire a Missile.
   vi. `GamePanel panel` – The custom `JPanel` to contain the GUI objects.

b. **Methods:**
   i. `GameFrame()` – This constructor is partially implemented. You are expected to layout and initialize other components.
   ii. Other methods in this class are already implemented for you.

Extra credit requirements (1-to-7 points)

There will be extra credit assigned based on:

a. substantive additional feature(s) added to the game, to create a unique game. The uniqueness can come from special rules, graphics, etc. If you are unsure, discuss your idea with your instructor.
b. You must briefly describe the added functionality along with one-or-more screenshot(s). documentation with screenshots as a PDF file titled `ExtraCreditDocumentation.pdf` and upload to Canvas.

Some ideas (you can come up with your own) for extra credit:

- You may want to change the rule set to make it more complicated or exciting (about 2-to-3 points).
- Maybe introducing key clicks or mouse clicks instead of the basic button at the bottom of the `JFrame` could be interesting (about 2-to-3 points)
- You can also investigate introducing image files and sound effects using a bit of additional research (about 3-to-4 points depending on results).
- Perhaps moving the entire turret left/right could be cool (about 4-to-5 points)
- **Coolest**: What would be the coolest if the turret's turret rotated (say ±60°) and missile's moved at that angle. (About 7 points). This idea is based on one of my favorite game, `Paratrooper` (https://www.youtube.com/watch?v=krb4ave2_Mg). I played the game on a 4 MHz (about 1000-times slower than your current computer) PC.

As long as you can shoot a missile at enemies and get a game over screen, you can be as creative as you want (this includes changing classes, adding classes, etc.)!

Testing for Phase #2:

There are no automated tests for Phase #2. It is up to you to test the operations of your game.

Phase #2: Submission via Canvas

This homework assignment must be turned-in electronically via Canvas CODE plug-in. Ensure your program compiles successfully, without any warnings or style errors. Ensure you have documented the methods with Javadoc. Upload the following to Canvas via the CODE plug-in:

1. The Java source files constituting your game.
2. Description of the extra credit functionality along with one-or-more screenshot(s) saved as a PDF file titled `ExtraCreditDocumentation.pdf`.