Paranoid Asteroid - Milestone 2

- Thomas Rahn (6286852)
- Michael Lavoie (9778004)
- Alan Ly (6293484)
- David Siekut (6329810)
- Dan Magariu (9217479)

Summary of Project

Paranoid Asteroid is a game written in Java, based on the classic arcade game, *Asteroid*. The game was written predominantly using the standard Java libraries. The only third-party libraries that were used, were for the purposes of unit testing and for MP3 playback. This project interests us in particular because it is written in Java (which all the group members are familiar with), it is a desktop end-user application, and being the result of a school project, there are likely to be improvements that could be made to the codebase.

Class Diagram of Actual System

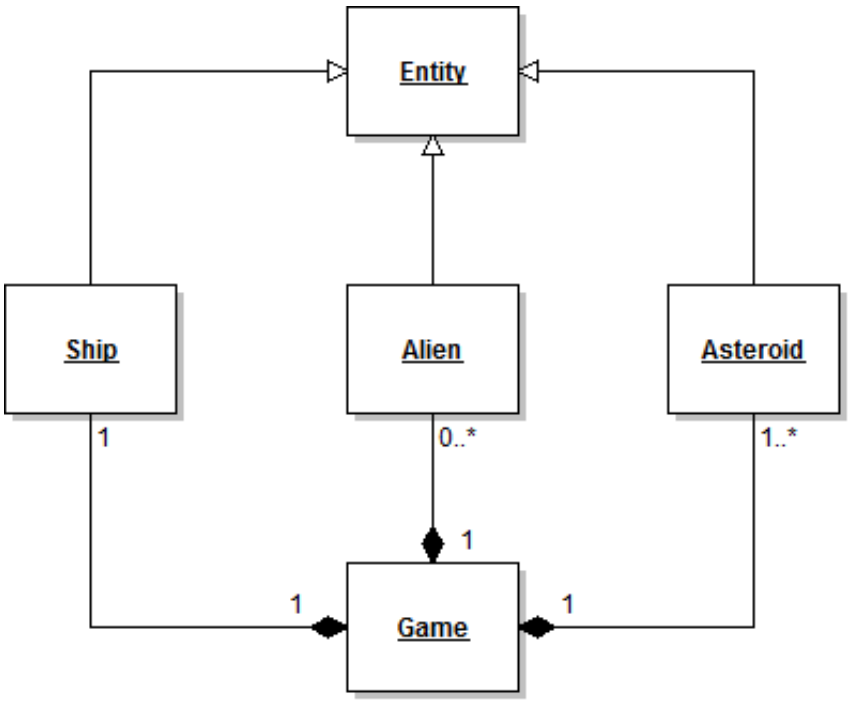


Figure 1 - Ideal Architecture

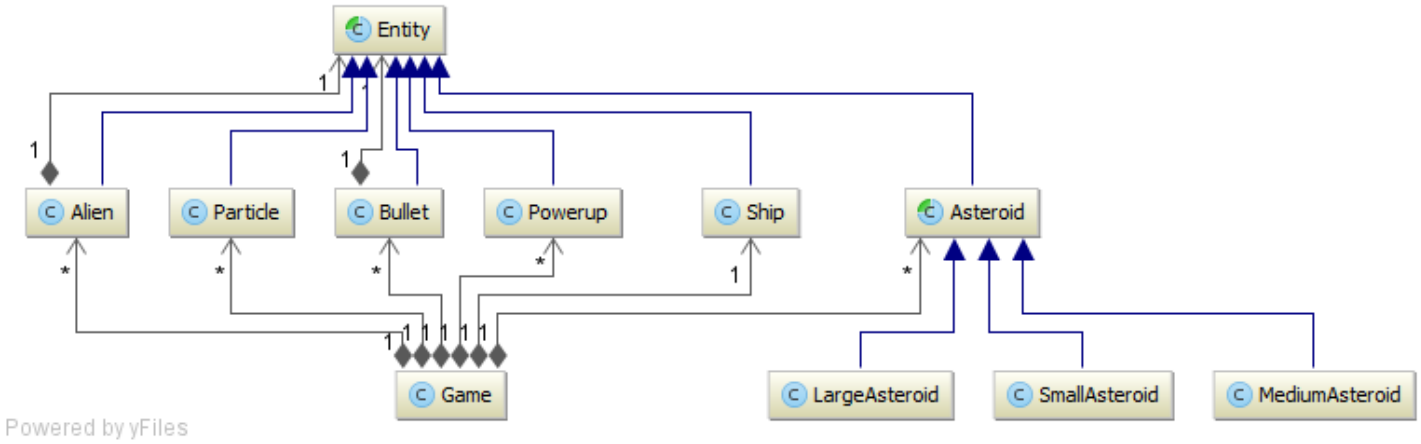


Figure 2 - Actual Architecture

Actual Architecture

There are 11 classes in the implementation.

We have the `Game` class which represents the state of a current game session and all it's different entities.

In terms of game entities, there is the `Entity` parent class containing the base implementation. Beneath that we have the `Alien`, `Particle`, `Bullet`, `Powerup`, `Ship`, and `Asteroid` classes. The `Game` class contains direct instances of those classes which all inherit from the `Entity` type.

For each entity, there is typically a one-to-zero-or-more relationship with the exception of `Ship`, which has a one-to-one relationship with the game as there is always only ever one user-controlled ship. As for the `Asteroid` class, there are three sub-types representing the available sizing of asteroids: small, medium, and large, and `Game` gets those instances of `Asteroid`.

Actual vs Conceptual

The actual architecture maps pretty well onto the conceptual architecture. We have the same layout of `Entity`, `Ship`, `Alien`, `Asteroid`, and `Game`. The difference in the actual implementation of the architecture is that there's a wider variety of entities and the implementation of asteroids is more particular.

This means that the original system is fairly well designed and didn't stray too far from the conceptual design.

Analysis Tools & Methodology

In order to better understand the actual architecture/implementation, the above UML relationship diagram was generated from the codebase. This was done using IntelliJ IDEA's bundled UML plugin.

Class Descriptions

Entities

```
public abstract class Entity

private Point center;
private Shape bounds;

public abstract void update(long delta);
public Point getCenter();
public Shape getBounds();
public void setCenter(Point center);
public void setBounds(Shape bounds);
```

Asteroid

```
public abstract class Asteroid extends Entity

public enum Size;
protected Point[] vertices;
protected Size size = Size.SMALL;
private double angle;
private double speed;

public static Asteroid buildAsteroid(Size size, Point center);
public void update(long delta);
protected abstract void initializeVertices();
private void updateVertices(long delta);
private void updateBounds();
```

Code Smells and Possible Refactoring

1. Duplicated Code

`game.GameController.java`

The *playSinglePlayer* and *playTwoPlayer* methods of the *GameController* class contain duplicated code. This can be easily seen in the *playTwoPlayer* method where there are groups of code that are the same except for minor differences. Copy and pasting is an indication of duplicated code and this is a good example because the code in question has duplicated comments that the author forgot to correct.

To fix this smell, I would use the **extract method** refactoring to remove similar lines of code and group them into their own method. In this case I would create a *playRound* method that creates a Game instance and initiates the game sequence. I would then remove the methods *playSinglePlayer* and *playTwoPlayer* and simply add a switch statement to the *playGame* method to deal with each *GameType*.

`game.Point.java`

In the `wrapAround` method, the following code block is repeated twice causing an unnecessarily long method that can otherwise be separated into its own method to avoid code duplication.

```
if (this.x < 0) {
    this.x += xMax;

    for (Point p : points) {
        p.x += xMax;
    }
} else if (this.x > xMax) {
    this.x -= xMax;

    for (Point p : points) {
        p.x -= xMax;
    }
}
```

2. Speculative Generality

`game.entities.Asteroid.java` `game.entities.SmallAsteroid.java`
`game.entities.LargeAsteroid.java`

Inheritance allows subclasses to inherit and extend the functionality of their parent classes. *SmallAsteroid* and *LargeAsteroid* merely change the initialization of the *Asteroid* class. They do not extend any behavior, nor do they use it. As their names suggest, *SmallAsteroid* and *LargeAsteroid* only define small and large asteroids respectively. Small and large are characteristics of an asteroid, not behavior. The UML below shows how useless these classes are.

I would use the **collapse hierarchy** refactoring method to remove the unnecessary *SmallAsteroid* and *LargeAsteroid* classes. The **pull-up method** method would then be used to migrate and merge the behavior from the removed classes to the *Asteroid* class.

3. Feature Envy

`game.Game.java`

asteroidCollided has feature envy (calls *Asteroid.getSize* once and *Asteroid.getCenter* twice), should be refactored using **Move Method** to *game.entities.Asteroid*.

applyPowerup has feature envy (calls *Ship.boost*, *Ship.arm*, *Ship.shield*, *Ship.pulseOn* each once), should be refactored using **Move Method** into *game.entitites.Ship*.

4. Poor Organization

`game.Game.java`

In the *applyPowerup* method, the following conditional block should be replaced by a **Strategy**.

```
if (type == Powerup.Power.BOOST) {
    ship.boost();
} else if (type == Powerup.Power.TRIPLE_SHOT) {
    ship.arm();
} else if (type == Powerup.Power.SHIELD) {
    ship.shield();
} else if (type == Powerup.Power.PULSE) {
    ship.pulseOn();
}
```

5. Large Class

`game.Game.java`

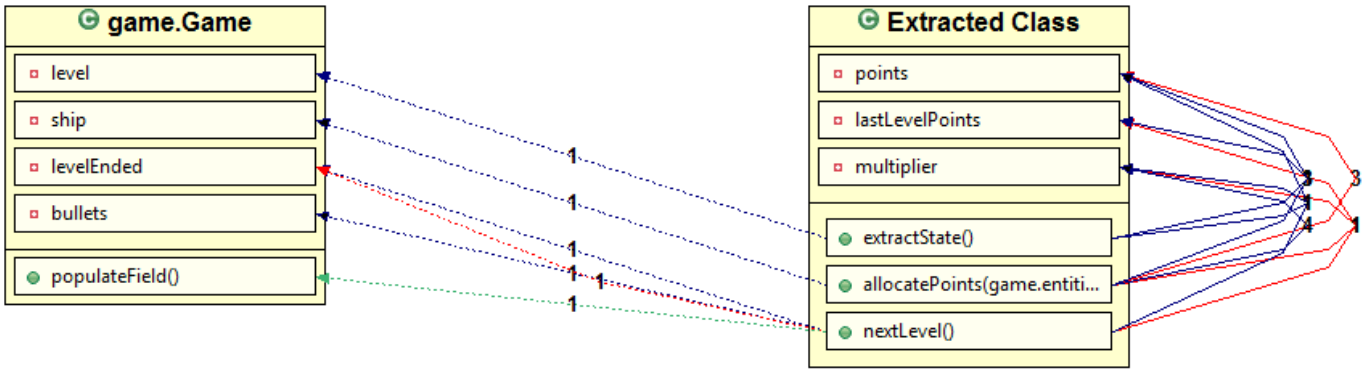
This class is very large (God class) and has several parts that can be refactored into another (new) class. One of these is the points system:

```
private void allocatePoints(Entity destroyed)
```

:

```
private long points;
private long lastLevelPoints;
private double multiplier;
```

This will move some of the instance variables and a method into another class, decoupling the Game class slightly and making it less complex.



6. Long Parameter List & Data Clumps

`game.entities.Bullet.java`

When creating a bullet, the **Bullet** constructor take in as parameters all three data item fields of the *BulletFiredEvent* object, which can be used together in lots of places.

```
new Bullet(e.getSource(), e.getOrigin(), e.getAngle());
```

In order to make the parameter list more robust and readable, I use the **Preserve Whole Object** refactoring on the Bullet constructor. I use the *BulletFiredEvent* as a constructor-argument and also change the event's name to remove ambiguity.

```
new Bullet(eBulletFired);
```

And so the Bullet constructor will be the following:

```
public Bullet(BulletFiredEvent eBulletFired) {
    setCenter(eBulletFired.center);
    this.source = eBulletFired.source;
    this.angle  = eBulletFired.angle;
    ...
}
```

Class Description

Bullet

```
public class Bullet extends Entity

private boolean expired;
private Entity source;
private double angle;
private double linearSpeed = 4.0e-7;
private long timeToLive = MAX_TIME_TO_LIVE;

public void update(long delta);
private void updateCenter(long delta);
private void updateBounds();
```