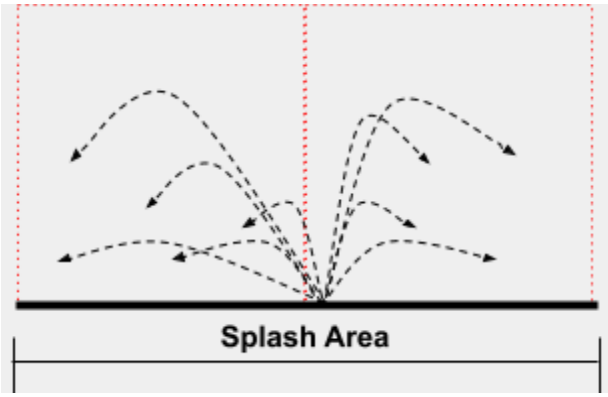
Case Study #7 - Exercise Solutions

Exercise #1

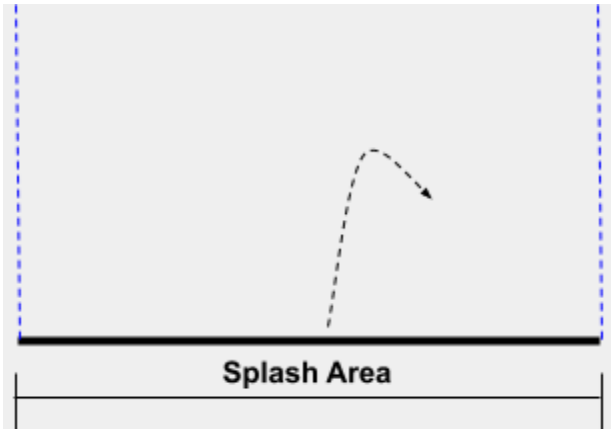
Assumptions:

1. We have to first define what we mean by “ground level” explosion: let's think of it this way:
    a. Assume that we have some sort of platform (i.e. ground) and we would like to explode a missile on it as if it was a bomb that was released by an air-plane for example. We could imagine something like this:

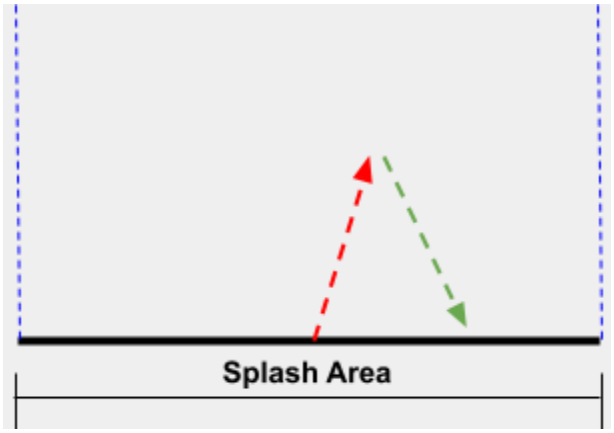


    b. The idea in the above illustration is as follows:
        i. Each particle that will be part of the simulation should only travel up and then down and land on the ground (or dissipate before it hits the ground) - it should not go lower than the ground level.
        ii. Each particle should not go further than the splash area (the the left or right of the start of the explosion)
        iii. Each particle should have a random trajectory made up of the following elements:
            1. It should go up and then let gravity bring it down.
            2. It should fly no further than the splash distance.
            3. It should be randomly generated to go into either the upper left or upper right quadrant (shown as squares with dotted red lines).
        iv. We will also make a simplification that all particles will travel at the same speed.

c. We can simplify this by concentrating on a single particle:

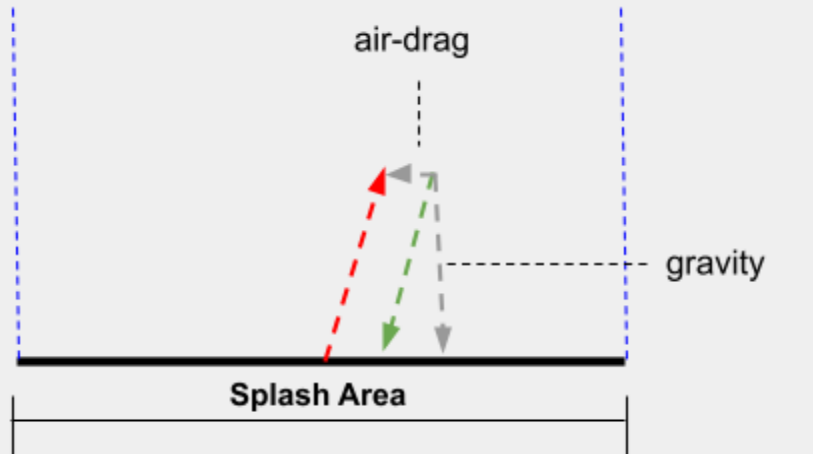


d. There are a few vectors here at play:



i. The simplest way to think about it is that we have the initial vector (red) and then we have the deceleration vector (green) which can be thought of as gravity and air resistance (for example `Vector2(-10, 130)` could simulate air resistance in the x direction with -10 going against a right moving particle and the 130 would be gravity pointing down.
ii. When the particle is moving to the left we should create an air-drag of +10 and when it's moving to the right we should set it at -10.
    1. We are basically going against the x-axis direction of the particle vector.
    2. In the y-axis we simply have a gravity vector pointing down at for example 130

e. A more accurate picture of the vectors would be as follows:



i. In the picture above we are showing how we would simulate the slowing down of the explosion particle by countering its movement along the x-axis with air-drag and its y-axis with gravity.

ii. In code we would define it as such:

```
static const gravity = 130.0;
static const airFriction = -10.0;
```

```
/// helper method to get the deceleration
/// portion of the particle path
static Vector2 getDeceleration(
      Vector2? originPosition, Vector2? initialSpeed) {
   Vector2 result =
      Vector2(ParticlesSystemExercise001.airFriction,
      ParticlesSystemExercise001.gravity);

   if (initialSpeed!.x < originPosition!.x) {
      result.x = -ParticlesSystemExercise001.airFriction;
   }
   return result;
}
```

f. Basically if we can simulate a single particle we can simulate them all. Lets first right down what the particle will accomplish:

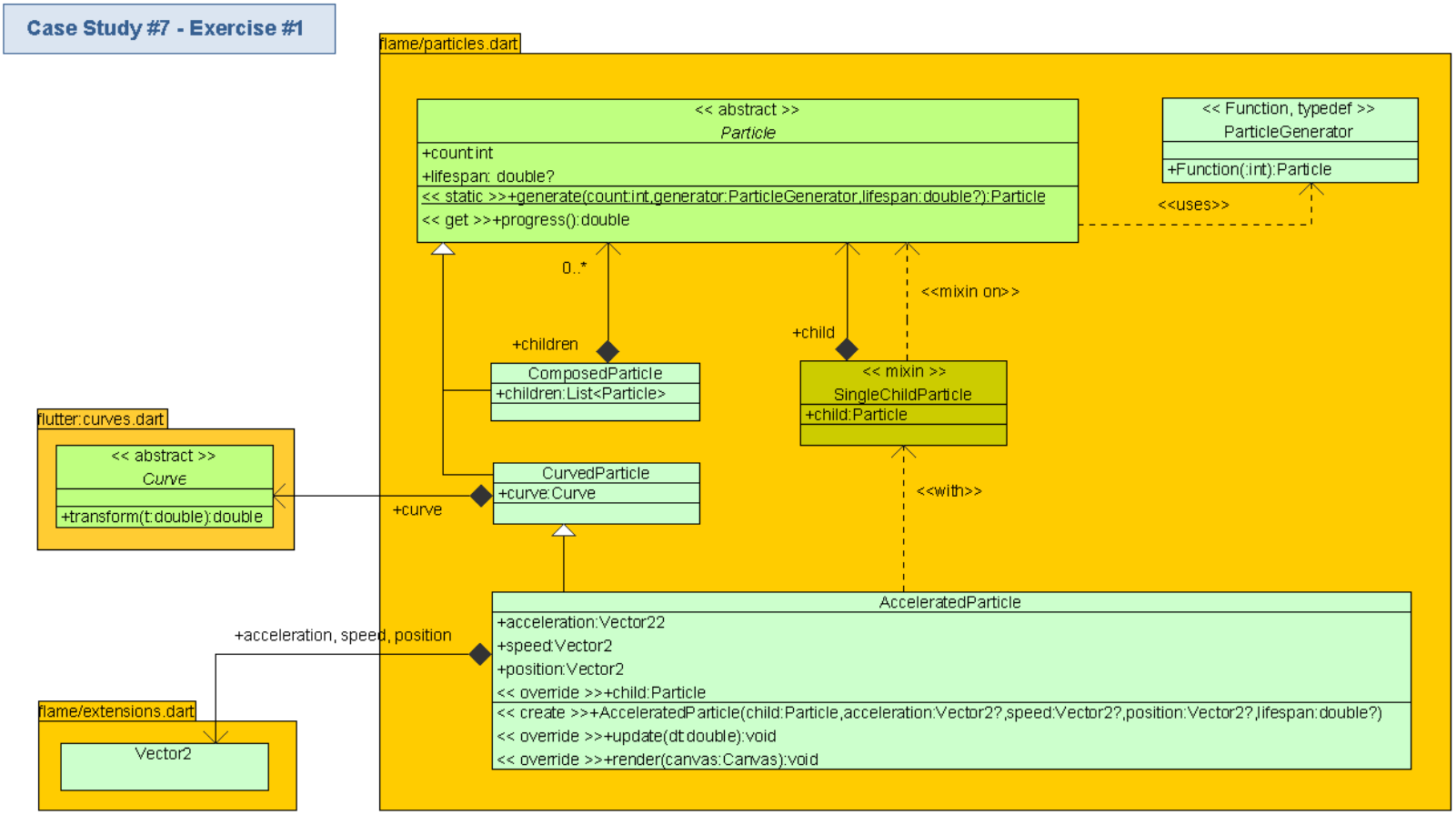
i. We can use **`AccceleratedParticle`** to simulate the acceleration and deceleration, we start with an acceleration and then constantly add gravity and wind resistance to counteract it so that eventually the direction of the

particle reverses and it falls to the ground to either left or right of the origin point.

ii. When the particle reaches the ground level. Which is simply saying:
    1. when the **`particle.position.y > origin.y`** we remove or make the particle invisible. (where **`origin`** is the position from which the explosion started.)

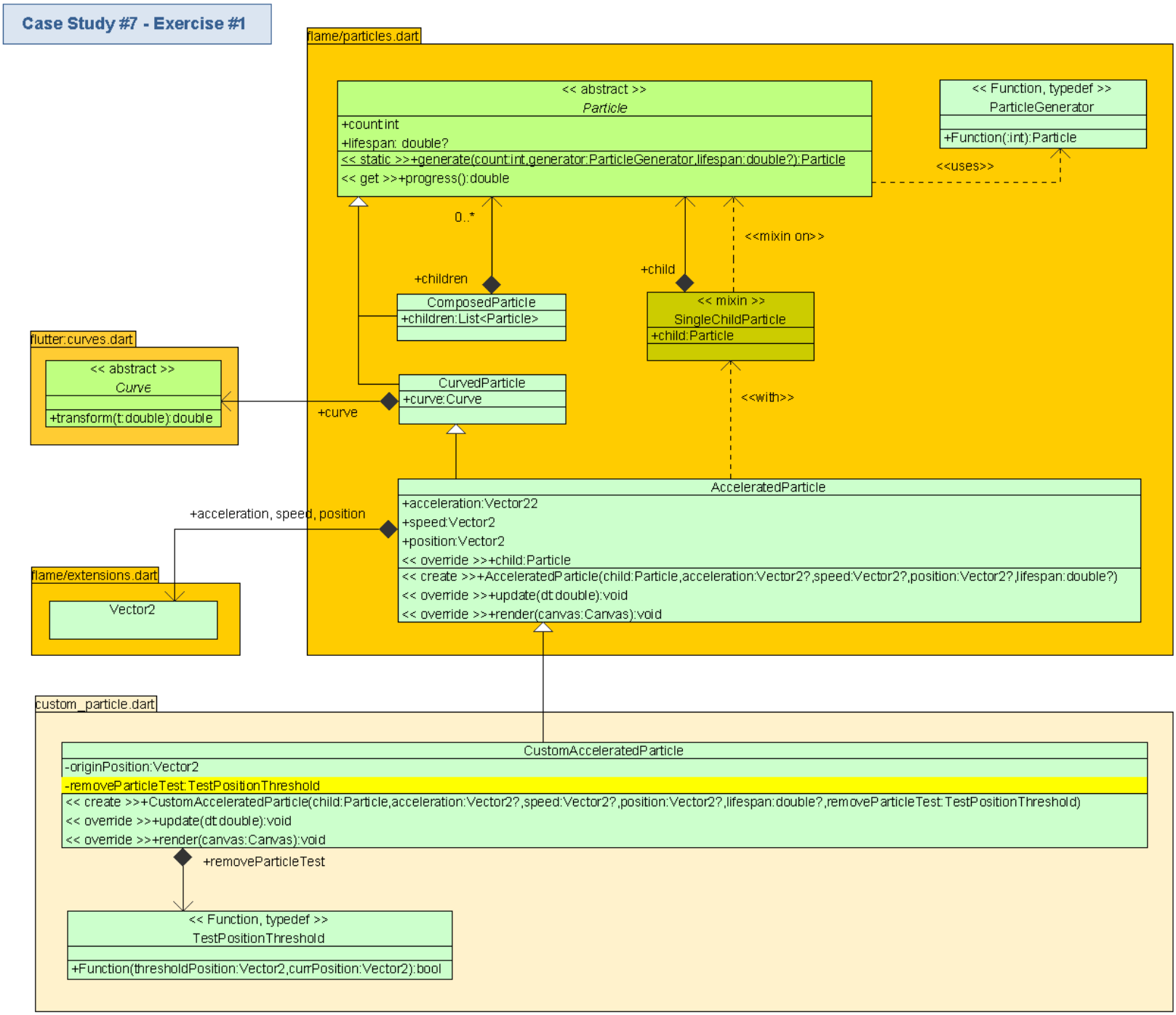
Solution:

1. The solution gets a bit complicated due to the following reasons:
    a. We need to know when a particle has gone lower than the origin point, which means that we have to have the ability to track individual particles and their positions.
    b. We need to create a way to generate random vectors (for directionality) to only fall in the two quadrants.
2. First let's look at how we would track the position of our particle. Look at this UML diagram of the **`AcceleratedParticle`**:



    a. You can see that **`AcceleratedParticle`** does indeed have a **`position`** variable that would allow us to see what the position of the particle is.
    b. So, what we need to do is somehow tell the particle that once its position is below the origin of the explosion it should stop rendering itself (i.e. disappear)
    c. Unfortunately the child particle (i.e. the rendering-particle) does not have proper access to the position of the parent particle, so what we will do is create a custom version of the **`AcceleratedParticle`** and basically test the position and once that position is below the origin we instruct our custom particle to stop rendering.
3. Here is the custom version of the particle:

a. Note the **`CustomAcceleratedParticle`** which allows us to intercept the position and control the update and render methods directly.
b. To allow for some level of customizability we added a **`TestPositionThreshold`** function which can be plugged into the particle to test the position threshold.



c. The test function will be as follows:

```
/// A function which returns a [bool] when called.
/// It accepts a [thresholdPosition] which specifies the position
/// to test against and [currPosition] which is the position we
/// are testing
typedef TestPositionThreshold = bool Function(
    Vector2 thresholdPosition, Vector2 currPosition);
```

d. And here is a simple implementation:

```
static bool removeParticleTest(
    Vector2 thresholdPosition, Vector2 currPosition) {
   if (currPosition.y > thresholdPosition.y) {
     return true;
   } else {
     return false;
   }
}
```

e. Then in the **`update`** and **`render`** methods of the **`CustomAcceleratedParticle,`** we simply check of the particle's position is below the threshold of the origin and we choke off the processing (i.e. we do not allow for the update and render of the particle to proceed thus making the particle disappear:

```
  void update(double dt) {
    if (!removeParticleTest(originPosition!, position)) {
      super.update(dt);
    } else {
      // do nothing
    }
  }
```

```
  void render(Canvas canvas) {
    if (!removeParticleTest(originPosition!, position)) {
      super.render(canvas);
    } else {
      // do nothing
    }
  }
```

4. Is there a better solution?

a. Yes
b. A better solution would be to create our own mixin that would allow us to create particles that broadcast their position to the caller. In essence it would be similar to the way the **`Particle`** class has the **`progress`** method which allows us to see the progress of the particle. We could do the same thing but for position.

You will find the code solution in the directory:
“Exercise Solutions\component_007[Particle_System]\exercise_007_001”

Exercise #2

Assumptions:

1. The code that we will be refactoring from will be coming from these sources:
    a. **\Exercise Solutions\component_005[Timers]\exercise_005_002**
        i. We will use it for the already existing Timer and LifeBar example and ball object generation.

    b. **\Exercise Solutions\component_003[Sound]\exercise_003_001**
        i. We will use it for spaceship, bullets, joystick control, and sound effects.

    c. **\Exercise Solutions\component_006[Parallax]\exercise_006_001**
        i. We will use it for additional sound effects as well as optional parallax effects..

    d. **\component_007[Particle_System]\sources\component_007_004**
    e. **\component_007[Particle_System]\sources\component_007_002**
        i. We will use the above use case-study code sources for explosions.

Solution:

1. We will use the code from the following case studies and exercises:
    a. **`\Exercise Solutions\component_005[Timers]\exercise_005_002`**: we will reuse the ball generation timer code to create our ball generator.
        i. We will modify this code as follows:
            1. Generate N balls at a specific interval (say every 4 seconds a new ball will be created - we can create for example 20 balls)
            2. We will remove the bouncing effects:
                a. Balls will not bounce off each other
                b. Balls will not bounce off the edges of the screen
            3. We will make all the balls 'wrap' around the edges of the screen so that when they reach beyond one edge they will reappear on the other side of the screen.

    b. **`\Exercise Solutions\component_003[Sound]\exercise_003_001`**: we will use the joystick, bullets, and sounds generated in the exercise.
        i. We will use **this code as our base code**. We will add all the other code into this code to create one code base.
            1. We will add the following files from Case-Study #5:
                a. `lifebar_text.dart`
                b. `my_collidable.dart`
            2. We will also make sure that the package for internationalization (i.e. intl: ^0.17.0) has been added in our pubspec.yaml
        ii. Then some modifications will be necessary:
            1. We have the following collisions we need to handle:
                a. Bullet object hits Ball object, Ball object hits Bullet object.
                b. Ball object hits SpaceShip object, SpaceShip object hits Ball object .
                c. **Note** that all the above collisions have the Ball objects in common so we can potentially handle all of the collisions in the **`OnCollision`** method of the **`MyColldable`** class. We do not need to check it in the other classes.
                d. We need to add to the **`Bullet`** and **`JoystickPlayer`** (i.e. the SpaceShip) classes the **`Collidable`** and **`HasHitboxes`** mixins.
                e. In order for the collisions to be detected we also must add a hitbox to both **`Bullet`** and **`JoystickPlayer`** using the **`addHitbox(...)`** method. We will in both cases just add approximating rectangles for our hitboxes.
            2. We will also modify the sound levels (i.e. volume) in the game:
                a. We are playing a total of a three sounds and music:
                    i. Background Music
                    ii. Bullet firing
                    iii. Bullet flying

iv. Explosion

b. What we would like to do is control the audio levels for all these sounds including music. In Flame we can use the range of 0 to 1 to set the volume level through the FlameAudio API.
c. We will set them as follows:
    i. Background Music (0.5)
    ii. Bullet firing (0.5)
    iii. Bullet flying (0.2)
    iv. Explosion (0.7)

c. \component_007[Particle_System]\sources\component_007_004 and **\component_007[Particle_System]\sources\component_007_002**: we will reuse the explosion code to add an explosion when a spaceship or a Ball object is destroyed:

i. When the Spaceship or a ball is destroyed we remove the objects and display the explosion in that location.
ii. We will use the particle explosion for the balls when they are destroyed and we will use the sprite animation explosion to show the spaceship exploding.

You will find the code solution in the directory:
"Exercise Solutions\component_007[Particle_System]\exercise_007_002"