Class Diagrams

Angle
-degrees
+toRadians()
+toDegrees()
+getDegrees(): degrees
+getRadians()
+setUp
+setDown
+setLeft
+setRight
+add()
-normalize()

Sentence: This class's convenience is easy, having no need of configuration to be used in this program. This class's fidelity could be considered extraneous if we don't use every method, but otherwise it's complete. And in abstraction it is complete as well, seeing that the client will never see or interact with the class.

Acceleration
- ddx:double
- ddy:double
+ Acceleration(dx, dy)
+ Acceleration(dx:double, dy:double)
+ getDX: double
+ getDY: double
+ getSpeed: double
+ setDX(dx:double): double
+ setDY(dy:double)
+ set(angle: Angle, magnitude:double)
+ addDX(dx: double)
+ addDY(dy: double)
+ add(rhs: Acceleration)

Sentence: This class is seamless for convenience because everything in this class is designed for this class. This class may be extraneous for fidelity because there are things from outside the class that are needed to complete the needs of the class. The class has complete abstraction because the class doesn't know the details of another class.

Velocity
- dx:double
- dy:double

+ Velocity()
+ Velocity(dx: double, dy: double)
+ getDX()
+ getDY()
+ getSpeed()
+ setDX(dx: double)
+ setDY(dy: double)
+ set(angle: Angle, magnitude: double)
+ addDX(dx:double)
+ addDY(dy:double)
+ add(acceleration: Acceleration)

Sentance: This class is seamless for convenience because everything in this class is designed for this class. This class may be extraneous for fidelity because there are things from outside the class that are needed to complete the needs of the class. The class has complete abstraction because the class doesn't know the details of another class.

Position
- X:double
- y:double
+ Position()
+ Position(pos:Position): x(pos.getX()), y(pos.getY())
+ Position(x:double, y:double)
+ getX()double return x
+ getY()double return y
+ Operator:bool (rhs:Position)
+ Operator:bool (rhs:Position)
+ setX(x:double)
+ setY(y:double)
+ addX(x:double)
+ addY(y:double)
+ add(a:Acceleration, v:Velocity, t:double)

Sentance: I think this class for convenience is easy because it covers everything it needs to accomplish this task. For fidelity I think this class is complete because it fulfils the requirements needed for this project. I think for abstraction it would be opaque because the class uses everything it needs for the class with knowing things from other classes.

Projectile
-mass
-distance
-angle: Angle
-acceleration: Acceleration
-status: Status

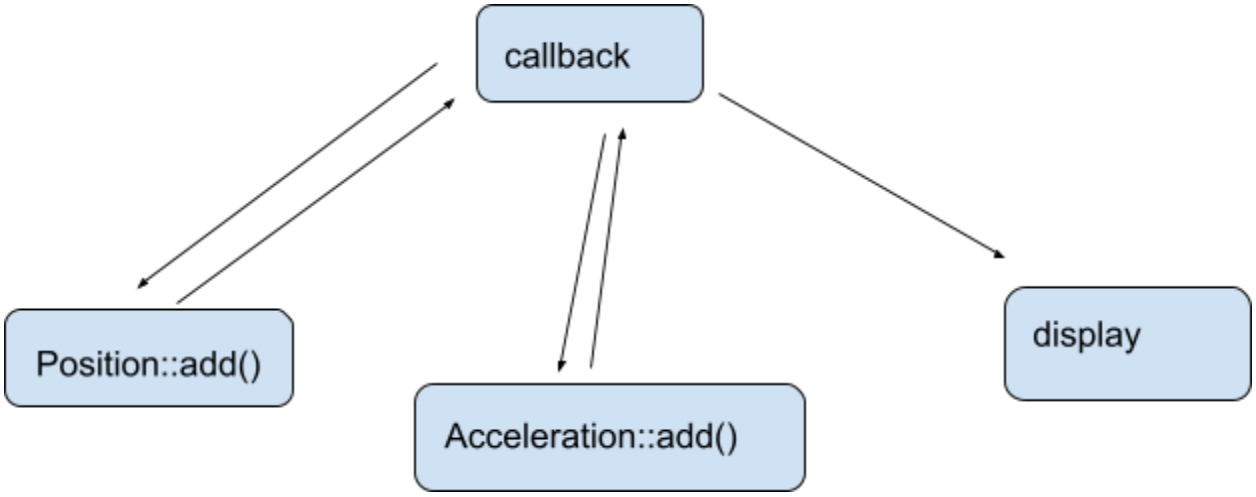
-velocity: Velocity
-pos: Position
+age
-calculateDistance()
+getDistance
+computeSpeed(dx, dy): velocity
+getPos(): pos
+getStatus(): status
+getAngleDeg(): angle.getDegrees

Sentence: This class is seamlessly convenient, because it takes a large load of logic out of the main program, thus being essential to it. This design has complete fidelity. It is exactly what the program needs. It also has complete abstraction. The user doesn't need understanding to use this. You plug it in, and it works in each case, at least on paper. We're going to write white box unit tests, making this robustness level proven.

Howitzer
-pos: Position
+angle: Angle
+fire()
+instantiateProjectile(projectilePrefab)
+handleInput()
+draw()
+reset()
+getPosition(): pos

Sentence: This class has a level of easy convenience. It takes a lot of logic off of the Main, but it needs to be called by the program and needs to be randomly set to a position, a method which would need to be imported in. It does have, though, complete fidelity. It's exactly what the program, and the projectile class, needs. The user doesn't need to know anything about it to use it, so it's a complete abstraction. I also want extensive white box unit tests for this class, making it proven robustness.

Structure Chart



Pseudo Code

Projectile:: update()

```
Put <- pos.Y() //Altitude
Put <- getAngle
totalVelocity <- computeTotalSpeed(dx,dy)
Put <- totalVelocity
If altitude > 0
        Put <- age //hangTime
Put <- getDistance()
```

Simulator:: callback()

```
howitzer <- Howitzer
ground <- Ground
howitzer.draw
ground.draw

If (projectile != null)
        projectile.update()
Else:
        post <- angle
        If (input == left or input == right)
                angle.add()
        If (input == tab)
                howitzer.fire()
        if (input == spacebar)
                ground.reset()
                howitzer.reset()
```