David Wells, Joe Allen, Josh Palmer

LAB 08 Artillery Design

UMLs:

Angle
- radians: double
+ setRadians(r: double): void
+ getRadians(): double
+ setDegrees(d: double); void
+ getDegrees(): double
+ normalize(): void

Fidelity: Complete

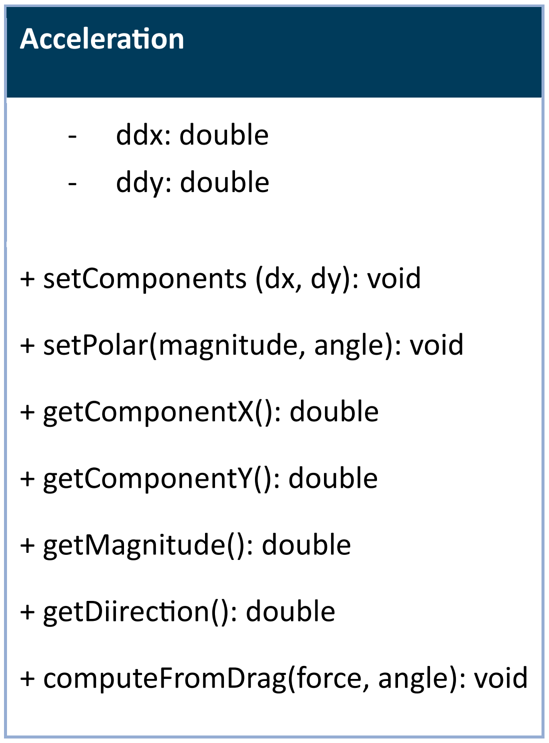
The class solely manages angle values, ensuring that the data is normalized and consistent. It performs no unrelated tasks.

Convenience: Seamless

This class is perfectly aligned with the needs of the application. It allows effortless conversion between degrees and radians and ensures angles remain normalized without requiring the client to manually adjust ranges. No additional work is necessary to manipulate or use this class.

Abstraction: Complete

The Angle class reveals no implementation details to the client. Conversion between degrees and radians, normalization logic, and internal storage format are fully encapsulated.



Acceleration

- ddx: double
- ddy: double

+ setComponents (dx, dy): void

+ setPolar(magnitude, angle): void

+ getComponentX(): double

+ getComponentY(): double

+ getMagnitude(): double

+ getDiirection(): double

+ computeFromDrag(force, angle): void

Fidelity: Complete

The class fully represents both component and derived vector state.

Convenience: Seamless or Easy

No extra work is needed—clients can set values in any format and retrieve meaningful summaries.

Abstraction: Opaque or Complete

Clients do not need to understand vector math.

Velocity
- dx: double
- dy: double
+ set(dx: double, dy double): void
+ getDX(): double
+ getDYY(): double
+ addAccel(ax: double, ay: double, dt: double): void
+ getSpeed(): double

Fidelity: Complete

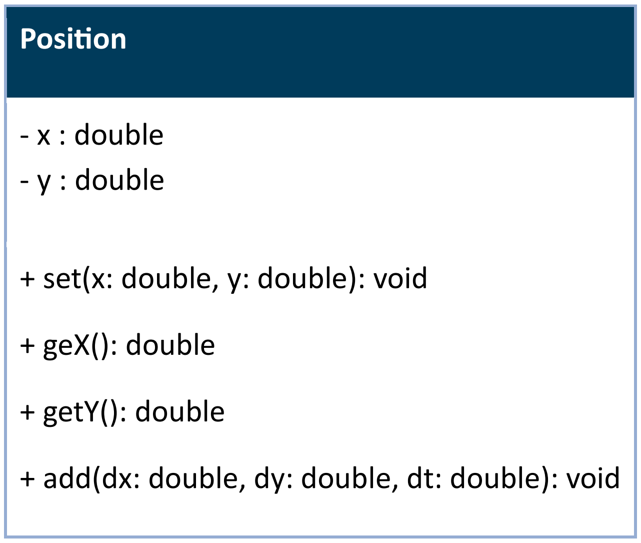
Velocity captures exactly the needed state (dx, dy) and includes operations to derive magnitude and update with acceleration.

Convenience: Easy

The Velocity class provides methods to get and update values and to apply acceleration in-place. Since speed calculation is built in and vector math is handled internally, no extra effort is required for most tasks.

Abstraction: Opaque

The client can manipulate velocity with simple interfaces, but unimportant implementation details like internal component separation (dx, dy) are still visible.


Position
- x : double
- y : double
+ set(x: double, y: double): void
+ geX(): double
+ getY(): double
+ add(dx: double, dy: double, dt: double): void

Fidelity: Complete

The class includes exactly the required fields (x, y) and behaviors related to object location.

Convenience: Seamless

The Position class integrates directly into simulation steps with methods that update its state based on simple velocity and time inputs.

Abstraction: Complete

The class hides all implementation details from the client. Position updates, storage, and coordinate manipulation are managed internally through clean interfaces. Clients use the class without needing to understand internal behavior.

Projectile
- pos: Position - vel: Velocity - acc: Acceleration - age: double - isFlying: bool
+ reset(): void + update(gravity: Gravity, air: AirDensity, drag: DragCoefficient, sos: SpeedOfSound): void + launch(howitzer): void + hasHitGround(): bool + getAltitude(): double + getDistance(): double + getSpeed(): double

Fidelity: Complete

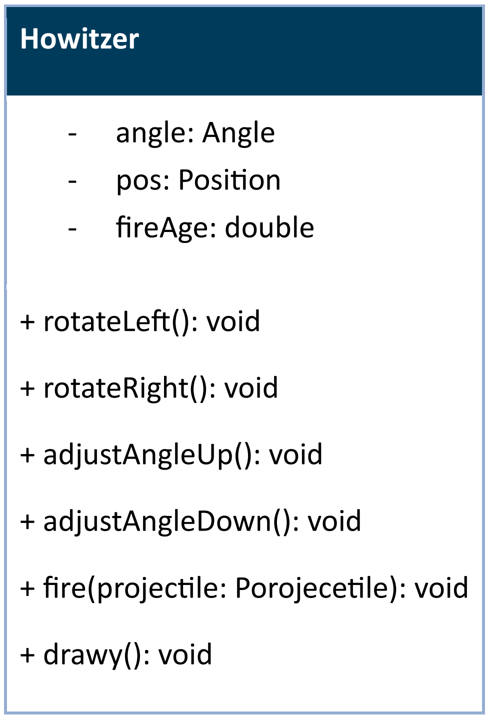
The class models the state of a projectile only

Convenience: Seamless or Easy

No tuning or tweaking needed externally.

Abstraction: Complete

Internal vectors and forces are hidden. Client only interacts through clear, meaningful methods.



Howitzer

- angle: Angle
- pos: Position
- fireAge: double

+ rotateLeft(): void

+ rotateRight(): void

+ adjustAngleUp(): void

+ adjustAngleDown(): void

+ fire(projectile: Porojecetile): void

+ drawy(): void

Fidelity: Complete

The Howitzer class includes position, angle, and fire state, perfectly representing the state and behavior of an artillery unit. It contains exactly what is needed to model and operate the howitzer in simulation.

Convenience: Seamless

The Howitzer class is directly tied to user input mappings.

Abstraction: Complete

The Howitzer class offers a high-level interface for rotation and firing, hiding all internal limits, position adjustments, and timing details. Clients can invoke behavior without needing any knowledge of the underlying implementation.

Pseudocode:

```
Method: update(gravity, airDensity, dragCoefficient, speedOfSound)

If projectile is not flying:
   Return

# 1. Get environmental values altitude
<- position.getY() g <-
gravity.get(altitude) ρ <-
airDensity.get(altitude) v <-
velocity.getSpeed() mach ← v /
speedOfSound.get(altitude)
c <- dragCoefficient.get(mach)

# 2. Compute drag force dragForce
<- 0.5 × c × ρ × $v^2$ × AREA dragAccel
<- dragForce / MASS
dragAngle <- velocity.getAngle()

# 3. Compute acceleration components acceleration.ddx
<- -dragAccel × sin(dragAngle)
acceleration.ddy <- -g - (dragAccel × cos(dragAngle))

# 4. Update position and velocity position.add(velocity.getDX(),
velocity.getDY(), TIME_STEP)
velocity.addAccel(acceleration.ddx, acceleration.ddy, TIME_STEP)

# 5. Update age
age <- age + TIME_STEP

# 6. Check for ground collision
If position.getY() ≤ 0:
isFlying <- false
```

Method: callback()

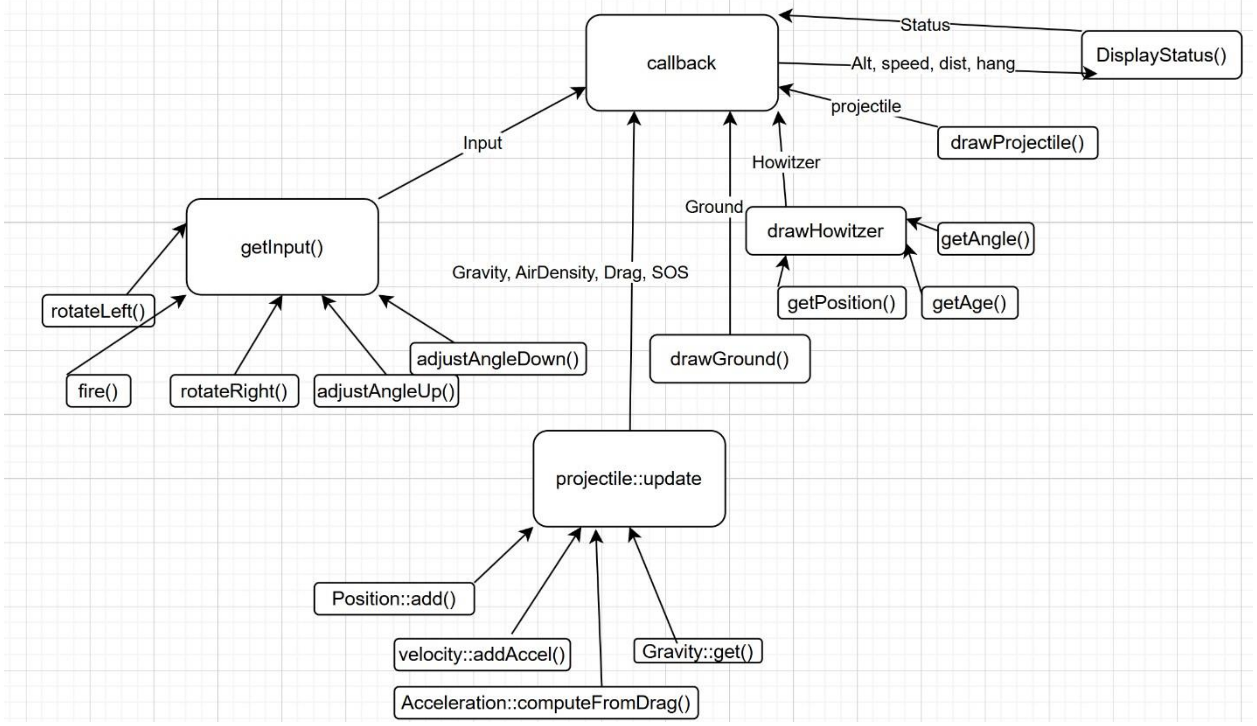
```
# 1. Handle user input If
left arrow pressed:
howitzer.rotateLeft() If
right arrow pressed:
howitzer.rotateRight() If
up arrow pressed:
   howitzer.adjustAngleUp() If
down arrow pressed:
   howitzer.adjustAngleDown()
If space bar pressed AND projectile is not flying:
   howitzer.fire(projectile)

# 2. Update projectile physics
projectile.update(gravity, airDensity, dragCoefficient, speedOfSound)

# 3. Draw all elements ground.draw() drawHowitzer(howitzer.getPosition(),
howitzer.getAngle(), howitzer.getAge()) If projectile is flying:
   drawProjectile(projectile.getPosition(), projectile.getAge())

# 4. Display status information If projectile is
flying:     display "Altitude:",
projectile.getAltitude()     display "Speed:",
projectile.getSpeed()     display "Distance:",
projectile.getDistance()     display "Hang
Time:", projectile.getAge() Else:
   display "Angle:", howitzer.getAngle().getDegrees()
```

Structure Chart


callback
Status
DisplayStatus()
Alt, speed, dist, hang
projectile
drawProjectile()
Input
Howitzer
getInput()
Ground
drawHowitzer
getAngle()
Gravity, AirDensity, Drag, SOS
getPosition()
getAge()
rotateLeft()
adjustAngleDown()
drawGround()
fire()
rotateRight()
adjustAngleUp()
projectile::update
Position::add()
velocity::addAccel()
Gravity::get()
Acceleration::computeFromDrag()