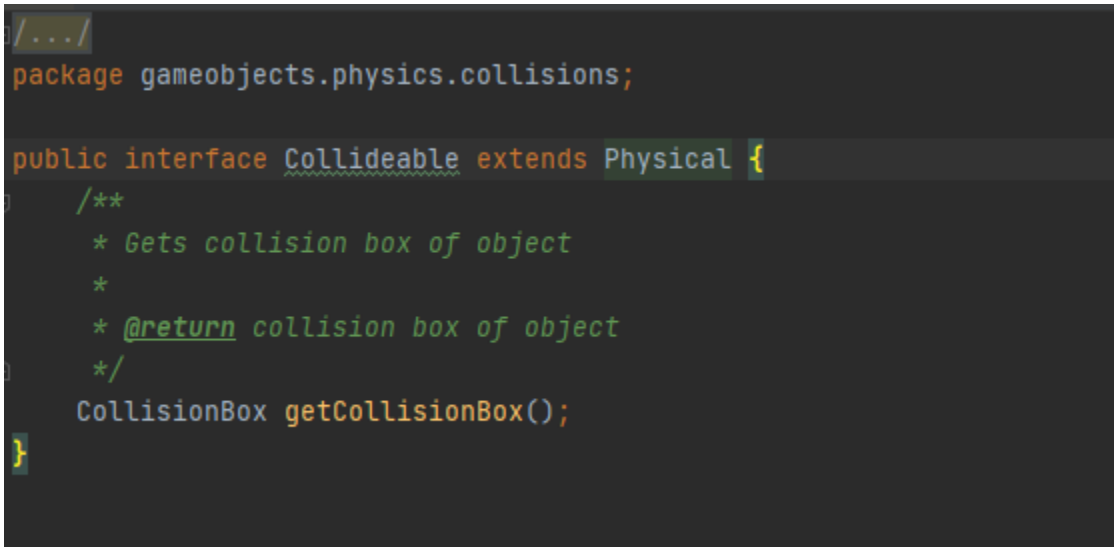
SOLID 1: Interface Segregation Principle

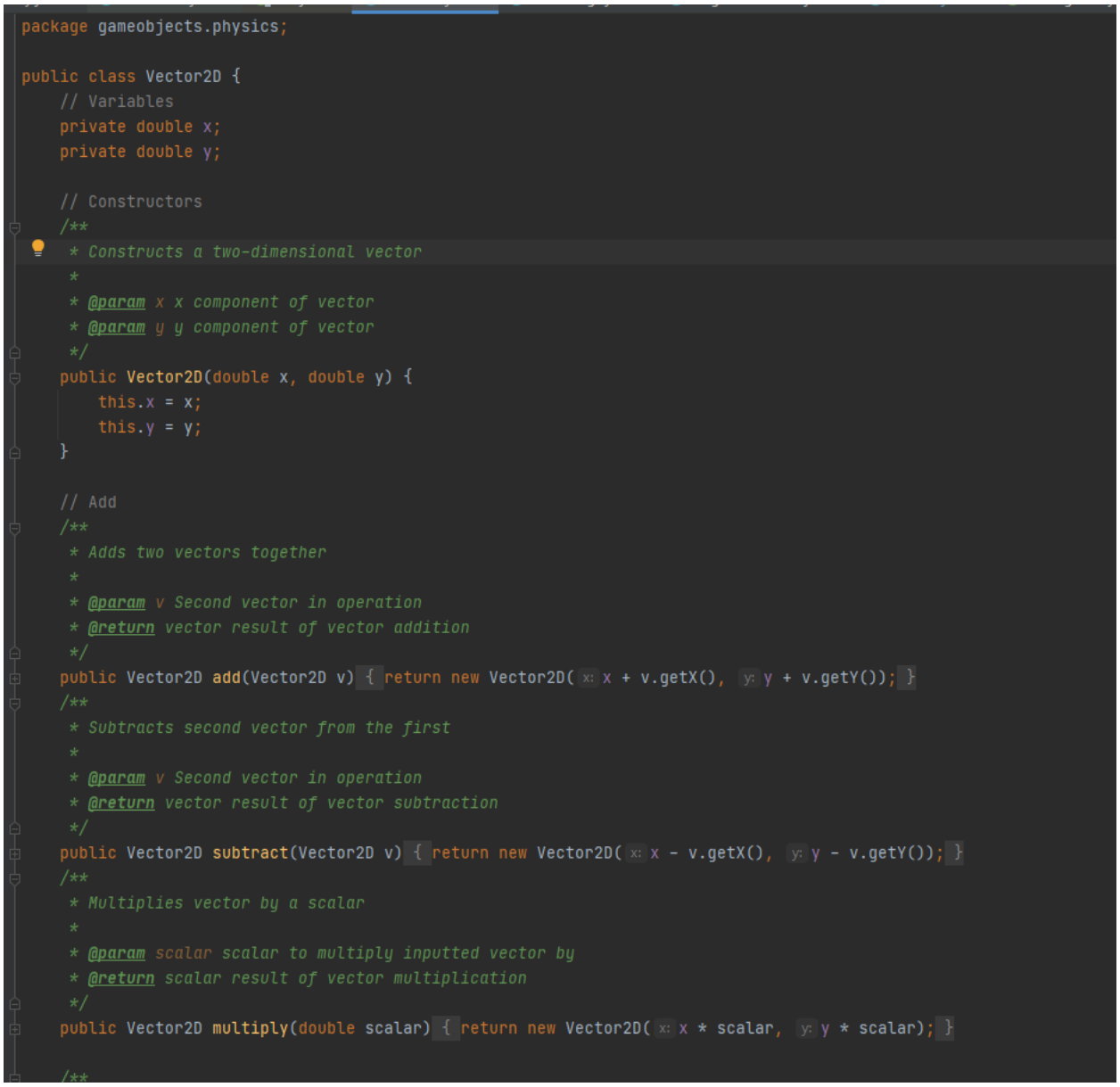


```
/.../
package gameobjects.physics.collisions;

public interface Collideable extends Physical {
    /**
     * Gets collision box of object
     *
     * @return collision box of object
     */
    CollisionBox getCollisionBox();
}
```

This interface is implemented by classes that require collision behavior. It is focused and simple, containing only a getCollisionBox() method. Any class can implement this interface and return a CollisionBox through getCollisionBox() to be able to be collided with. Classes that don't require collisions do not have to implement this interface, so the client doesn't have to depend on anything they do not need.

SOLID 2: Single Responsibility Principle



```java
package gameobjects.physics;

public class Vector2D {
    // Variables
    private double x;
    private double y;

    // Constructors
    /**
     * Constructs a two-dimensional vector
     *
     * @param x x component of vector
     * @param y y component of vector
     */
    public Vector2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Add
    /**
     * Adds two vectors together
     *
     * @param v Second vector in operation
     * @return vector result of vector addition
     */
    public Vector2D add(Vector2D v) { return new Vector2D( x: x + v.getX(),  y: y + v.getY()); }
    /**
     * Subtracts second vector from the first
     *
     * @param v Second vector in operation
     * @return vector result of vector subtraction
     */
    public Vector2D subtract(Vector2D v) { return new Vector2D( x: x - v.getX(),  y: y - v.getY()); }
    /**
     * Multiplies vector by a scalar
     *
     * @param scalar scalar to multiply inputted vector by
     * @return scalar result of vector multiplication
     */
    public Vector2D multiply(double scalar) { return new Vector2D( x: x * scalar,  y: y * scalar); }

    /**
```

```java
public Vector2D subtract(Vector2D v) { return new Vector2D( x: x - v.getX(),  y: y - v.getY()); }
/**
 * Multiplies vector by a scalar
 *
 * @param scalar scalar to multiply inputted vector by
 * @return scalar result of vector multiplication
 */
public Vector2D multiply(double scalar) { return new Vector2D( x: x * scalar,  y: y * scalar); }

/**
 * Multiplies a vector by -1 (a.k.a. flips it, etc.)
 *
 * @return vector result of operation
 */
public Vector2D opposite() { return multiply(-1); }
/**
 * Solves for the length of a vector
 *
 * @return scalar result of operation
 */
public double len() { return Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2)); }
/**
 * Solves for the squared length of a vector
 * Allows for a dot product optimization
 *
 * @return scalar result of operation
 */
public double squaredLen() { return x * x + y * y; }
public Vector2D round() { return new Vector2D(Math.round(x), Math.round(y)); }
public Vector2D roundDown() { return new Vector2D(Math.floor(x), Math.floor(y)); }
public Vector2D roundUp() { return new Vector2D(Math.ceil(x), Math.ceil(y)); }


/**
 * Normalizes a vector (finds its unit vector)
 *
 * @return the unit vector
 */
public Vector2D norm() {
    double len = len();

    if (len > 0) {
        return new Vector2D( x: x / len,  y: y / len);
    } else {
        return new Vector2D( x: 0,  y: 0);
    }
```

This class represents a two-dimensional vector which is able to be manipulated through basic operations. From the code above, the Vector2D class contains an add(Vector2D), subtract(Vector2D), multiply(double), opposite(), len(), squaredLen(), and norm() method, as

well as several others. The only reason this class would change would be if there was an error in vector arithmetic. Depending on the “Vector2D” abstraction while keeping the SRP in mind allows the reader to know immediately what the method does rather than waste time digging through a code base trying to find related code.

SOLID 3. Open-Closed Principle

```java
/.../
package gameobjects;

import ...

public abstract class GameObject implements Physical, Drawable {
    // Variables
    private PhysicsControllerRelative physics;
    private SpriteController graphics;
    private ImageSheet sheet;
    private Room room;
    private Vector2D center;

    // Constructors
    public GameObject(Room room, double initialX, double initialY,
                      double centerX, double centerY, ImageSheet sheet) {
        this.physics = new PhysicsControllerRelative(initialX, initialY, room.getPhysics());
        Sprite sprite = new Sprite((int) initialX,
                (int) initialY, (int) (centerX * 2),
                (int) (centerY * 2), Main.WALLTILE); // walltile = default no texture
        this.room = room;
        this.graphics = new SpriteController(sprite, sheet.getInitialReel());
        this.center = new Vector2D(centerX, centerY);
        this.sheet = sheet;
    }

    public void update(Camera camera) { update(camera, Main.DEFAULT_FRICTIONAL_FORCE); }

    public void update(Camera camera, double frictionalForce) {
        physics.update(frictionalForce);
        graphics.getSprite().setTranslateX(
                physics.getPosition().getX() - camera.getPhysics().getPosition().getX()
                + camera.getOffsetX() - center.getX());
        graphics.getSprite().setTranslateY(
                physics.getPosition().getY() - camera.getPhysics().getPosition().getY()
                + camera.getOffsetY() - center.getY());
    }

    // Getters
    @Override
    public PhysicsController getPhysics() { return physics; }
    public PhysicsControllerRelative getPhysicsRel() { return physics; }
    @Override
    public SpriteController getGraphics() { return graphics; }
    @Override
    public ImageSheet getSpriteSheet() {
        return sheet;
    }

}
```

Extending from the class allows children to represent physical objects that are drawn within the game. Since this class is abstract, programmers are forced to extend from this class, adding new

behavior through extension rather than modification. Methods such as update(Camera, frictionalForce) can also be overridden to change existing behavior.

GRASP 1: Controller

```java
/.../

package gameobjects.graphics.functionality;

import ...

public class SpriteController {
    // Variables
    private Sprite sprite;
    private ImageReel reel;
    private static Timer animationTimer = new Timer();

    // Constructors
    public SpriteController(Sprite editSprite, ImageReel initialReel) {
        this.reel = initialReel;
        this.sprite = editSprite;
        sprite.setImage(initialReel.getInitialImage());
        startAnimation();
    }

    // Misc.
    public void startAnimation() { animationTimer.schedule(nextAnimationFrame, delay: 100, period: 100); }
    private TimerTask nextAnimationFrame = () -> { sprite.setImage(reel.getNextImage()); };

    // Getters
    public Sprite getSprite() { return sprite; }
    public ImageReel getCurrentReel() { return reel; }

    // Setters
    public void setCurrentReel(ImageReel reel) {
        if (reel != this.reel) {
            this.reel = reel;
            sprite.setImage(reel.getInitialImage());
        }
    }
}
```

The SpriteController class coordinates the Sprite, ImageReel, Timer, and TimerTask classes to create an animation. It can switch between ImageReels (change the animation being played), start and stop the animation, and get the instance of the sprite it is drawing to. The class does

very little work on its own, only starting animation on construction and setting the delay between animation frames. It instead relies on the behaviors of the classes it coordinates.

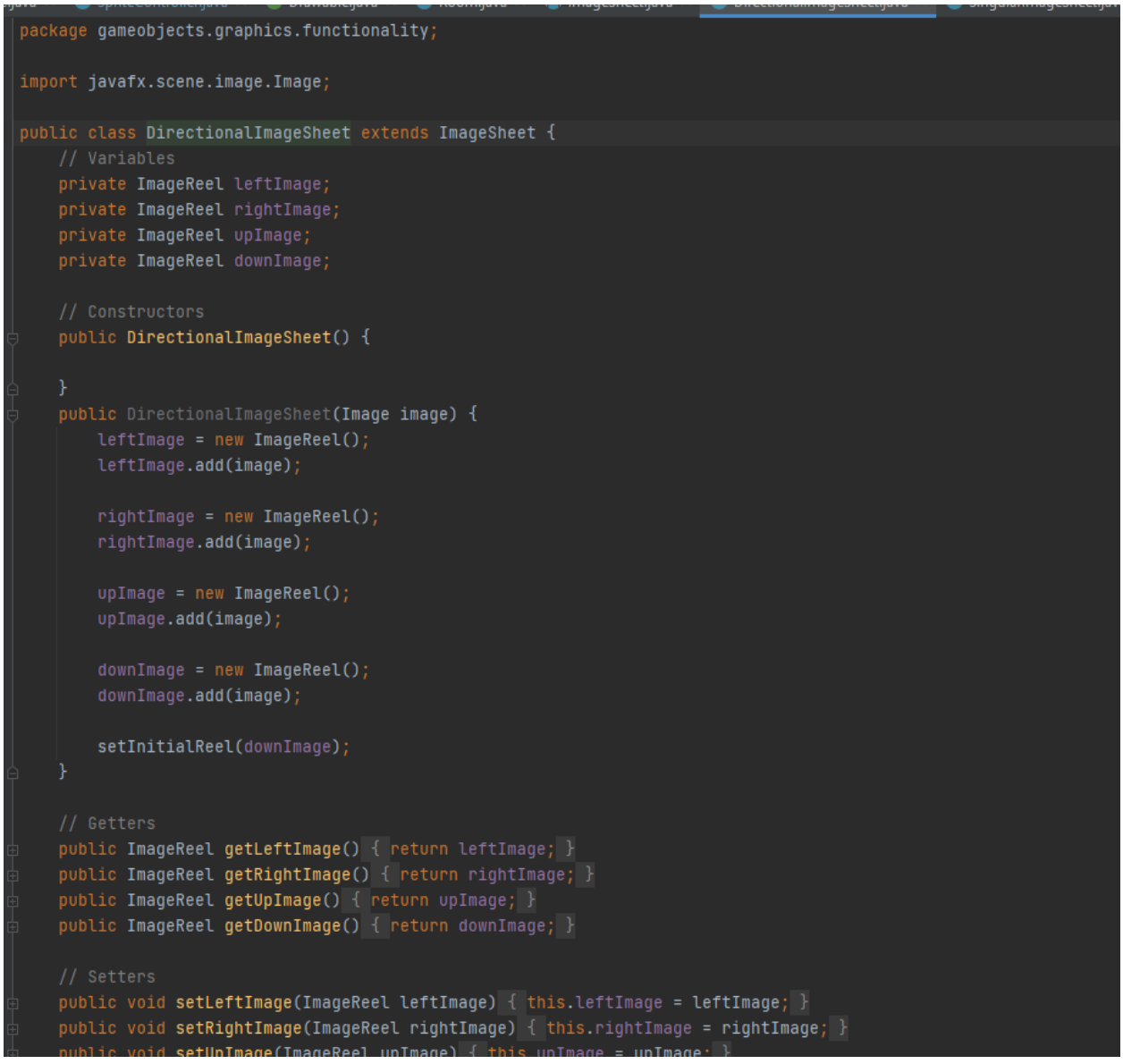
GRASP 2: Creator

```
public void finalize(Pane pane) {
    addRoomLayout();
    addChest();
    addNPC();
    generateMonsters();
    addFloorTiles();
    addSurroundingWalls();
    addAllSprites(pane);
    for (Monster monster: monsters) {
        monster.addHPBar( room: this, pane);
    }
}
private void addFloorTiles() {
    for (int r = 0; r < width; r++) {
        for (int c = 0; c < height; c++) {
            Tile tile = new FloorTile( inRoom: this, r, c);
            add(tile);
        }
    }
}
private void addSurroundingWalls() {
    // Add walls around the dungeon
    for (int r = 0; r < width; r++) {
        Tile topWall = new WallTile( inRoom: this, r,  initialY: -1);
        Tile bottomWall = new WallTile( inRoom: this, r, height);
        add(topWall);
        add(bottomWall);
    }
```

These methods are contained within the Room class. Upon a room's creation, finalize() places walls within the room, adds a chest, adds an NPC, generates monsters, adds floor tiles, and generates the sprites of the game objects within the room. Since all game objects are tracked

within the Room, the class closely uses the game objects that are created (NPC, Tile, etc.), making it a good candidate for the Creator GRASP principle.

GRASP 3: Pure Fabrication



```java
package gameobjects.graphics.functionality;

import javafx.scene.image.Image;

public class DirectionalImageSheet extends ImageSheet {
    // Variables
    private ImageReel leftImage;
    private ImageReel rightImage;
    private ImageReel upImage;
    private ImageReel downImage;

    // Constructors
    public DirectionalImageSheet() {

    }
    public DirectionalImageSheet(Image image) {
        leftImage = new ImageReel();
        leftImage.add(image);

        rightImage = new ImageReel();
        rightImage.add(image);

        upImage = new ImageReel();
        upImage.add(image);

        downImage = new ImageReel();
        downImage.add(image);

        setInitialReel(downImage);
    }

    // Getters
    public ImageReel getLeftImage() { return leftImage; }
    public ImageReel getRightImage() { return rightImage; }
    public ImageReel getUpImage() { return upImage; }
    public ImageReel getDownImage() { return downImage; }

    // Setters
    public void setLeftImage(ImageReel leftImage) { this.leftImage = leftImage; }
    public void setRightImage(ImageReel rightImage) { this.rightImage = rightImage; }
    public void setUpImage(ImageReel upImage) { this.upImage = upImage; }
```

We ended up needing code that could change a sprite to represent an action in four directions. For instance, walking north, south, east, and west should be modelled by the code. We created a

class called DirectionalImageSheet that takes four ImageReels (sprites with multiple frames) which can be gotten/set accordingly. Instances of this class can be passed into the Monster class constructor to ensure the Monster has a new sprite in multiple directions without adding code to the Monster class, for instance. We had to invent this class to avoid duplicate code and to increase maintainability.

GRASP 4: Protected Variations (Abstraction)

```
public abstract class Potion implements Consumable, Comparable {

    private ImageSheet spriteSheet;
    private String name;
    private int value;
    public Potion(String name, int value, ImageSheet spriteSheet) {
        this.spriteSheet = spriteSheet;
        this.name = name;
        this.value = value;
    }
    /**
     * Abstract implementation of consume from Consumable interface.
     * Will be overridden by each of the other potion
     * classes.
     * @param player player object that consumes the potion
     */
    public abstract void consume(Player player);

    /**
     * Abstract method to get the name of the potion.
     * @return name of the potion
     */
    public String getName() {
        return name;
    }

    public void setValue(int value) { this.value = value; }

    public int getValue() { return value; }
    public ImageSheet getImage() { return this.spriteSheet; }
```

The potion class is an abstract class that can be inherited from to create different types of potions. Potions should implement their own consume(player) methods, as they lead to different behaviors once drunk. This protects variations in child classes from affecting other potions.

GRASP 5: Indirection

```
/**
 * shoot bullet
 * @param room the room player is in
 * @param pane the overall pane
 * @param camera the camera
 * @param monsters list of monster
 */
public void launchProjectile(Room room, Pane pane,
                             Camera camera, LinkedList<Monster> monsters) {

    if (weaponList.size() < 0 || weaponList.get(holdingWeapon) == null) {
        ProjectileLauncher weapon = ProjectileLauncherA.getInstance(this);
        weaponList.add(weapon);
        holdingWeapon = weaponList.size() - 1;
    }
    ProjectileLauncher weapon = weaponList.get(holdingWeapon);
    weapon.shoot(room, pane, camera);
}
```

We found ourselves copy/pasting functionality to shoot projectiles. We fixed this by creating a ProjectileLauncher class to decouple the Player, Monster, etc. class shooting the projectile with the Projectile class. Coupling is therefore reduced and it is easier to reuse functionality. While it should probably not be a singleton and should be tied directly to the instance during its creation, this is an example of indirection.