SOLID Design Principles

Instructor: Mehroze Khan

SOLID Principles

- Good software systems begin with clean code
- The SOLID principles tell us how to **arrange our functions and data structures** into classes, and how those classes should be **interconnected**
- The goal of the principles is the creation of mid-level software structures that:
    - Tolerate change
    - Are easy to understand
    - Are the basis of components that can be used in many software systems
- The term “mid-level” refers to the fact that these principles are applied by programmers working at the module level.

SOLID Principles

- **<u>S</u>RP:** Single Responsibility Principle
- **<u>O</u>CP:** Open-Closed Principle
- **<u>L</u>SP:** Liskov Substitution Principle
- **<u>I</u>SP:** Interface Segregation Principle
- **<u>D</u>IP:** Dependency Inversion Principle

Aik new interface bana do or sari functionalities + classes uske sath attach kardo so that aik new functionality attach ho jaye or old ko change na karna paray

OCP-Open Closed Principle

- The Open-Closed Principle requires that **classes should be open for extension and closed to modification.**
- Modification means **changing the code** of an existing class, and extension means **adding new functionality**.
- We should be able to add new functionality without touching the existing code for the class.
- This is because whenever we modify the existing code, we are taking the **risk of creating potential bugs**.
- We should avoid touching the tested and reliable (mostly) production code if possible.

OCP-Open Closed Principle

- But how are we going to add new functionality without touching the class?
- It is usually done with the help of **interfaces** and **abstract classes**.

```
// Violating OCP
class AreaCalculator {
public:
    double calculateArea(Shape shape) {
        if (shape.type == "circle") {
            return 3.14 * shape.radius * shape.radius;
        } else if (shape.type == "square") {
            return shape.side * shape.side;
        }
        return 0;
    }
};

// Following OCP
class Shape {
public:
    virtual double area() = 0; // Abstract method
};

class Circle : public Shape {
public:
    double radius;
    double area() override { return 3.14 * radius * radius; }
};

class Square : public Shape {
public:
    double side;
    double area() override { return side * side; }
};

class AreaCalculator {
public:
    double calculateArea(Shape* shape) {
        return shape->area();
    }
};
```

Example-1

- Let's say our boss came to us and said that they want invoices to be saved to a database so that we can search them easily.
- We think okay, this is easy peasy boss, just give me a second!
- We create the database, connect to it, and we add a save method to our **InvoicePersistence** class

```
public class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }

    public void saveToDatabase() {
        // Saves the invoice to database
    }
}
```

- We as the lazy developer for the bookstore, did not design the classes to be easily extendable in the future. So, to add this feature, we have modified the **InvoicePersistence** class.
- If our class design obeyed the Open-Closed principle, we would not need to change this class.
- So, as the lazy but clever developer for the bookstore, we see the design problem and decide to refactor the code to obey the principle.

How to Fix?

```
interface InvoicePersistence {
    public void save(Invoice invoice);
}
```

- We change the type of **InvoicePersistence** to Interface and add a save method. Each persistence class will implement this save method.

How to Fix?

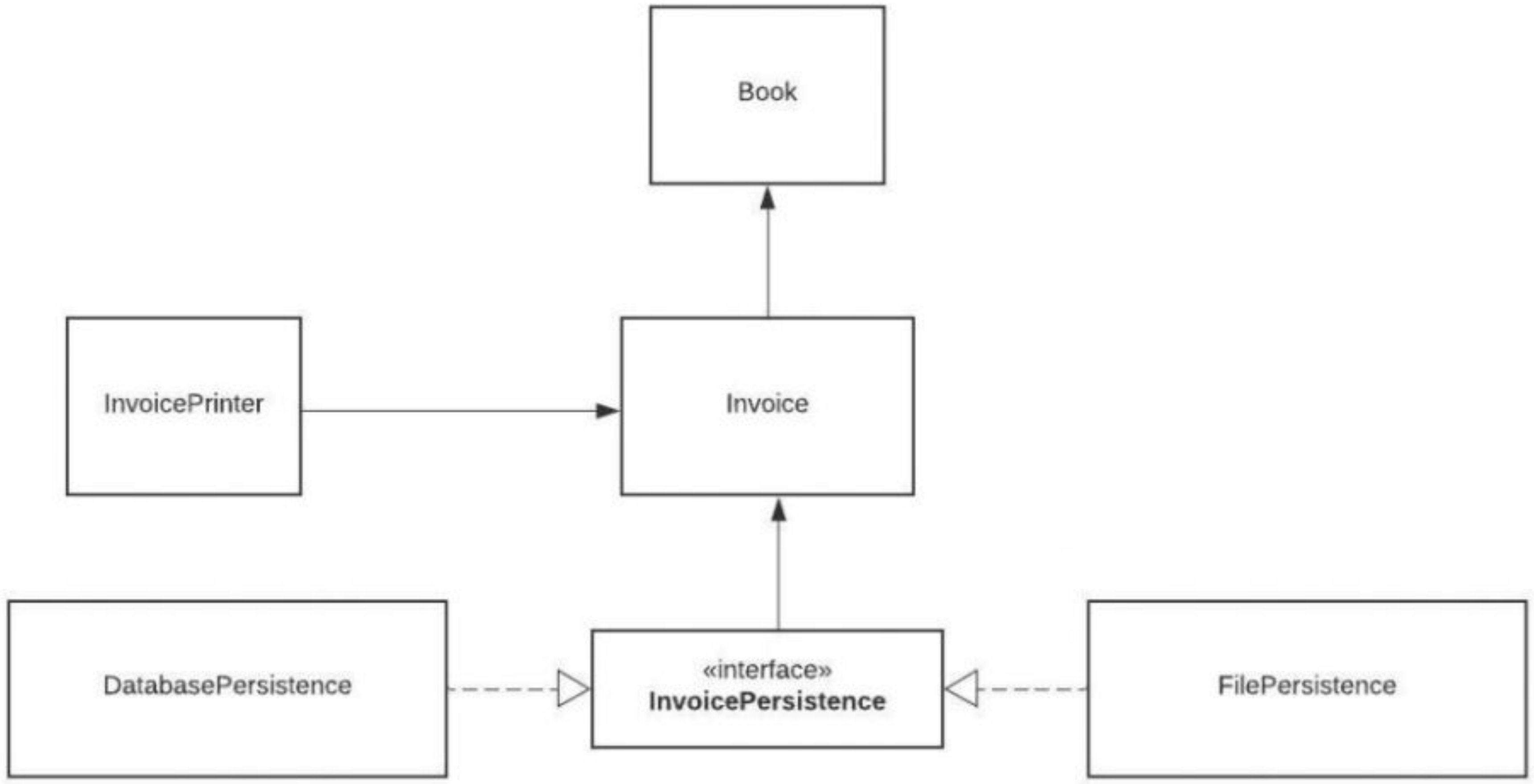
```
public class DatabasePersistence implements InvoicePersistence {
    // Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}
```

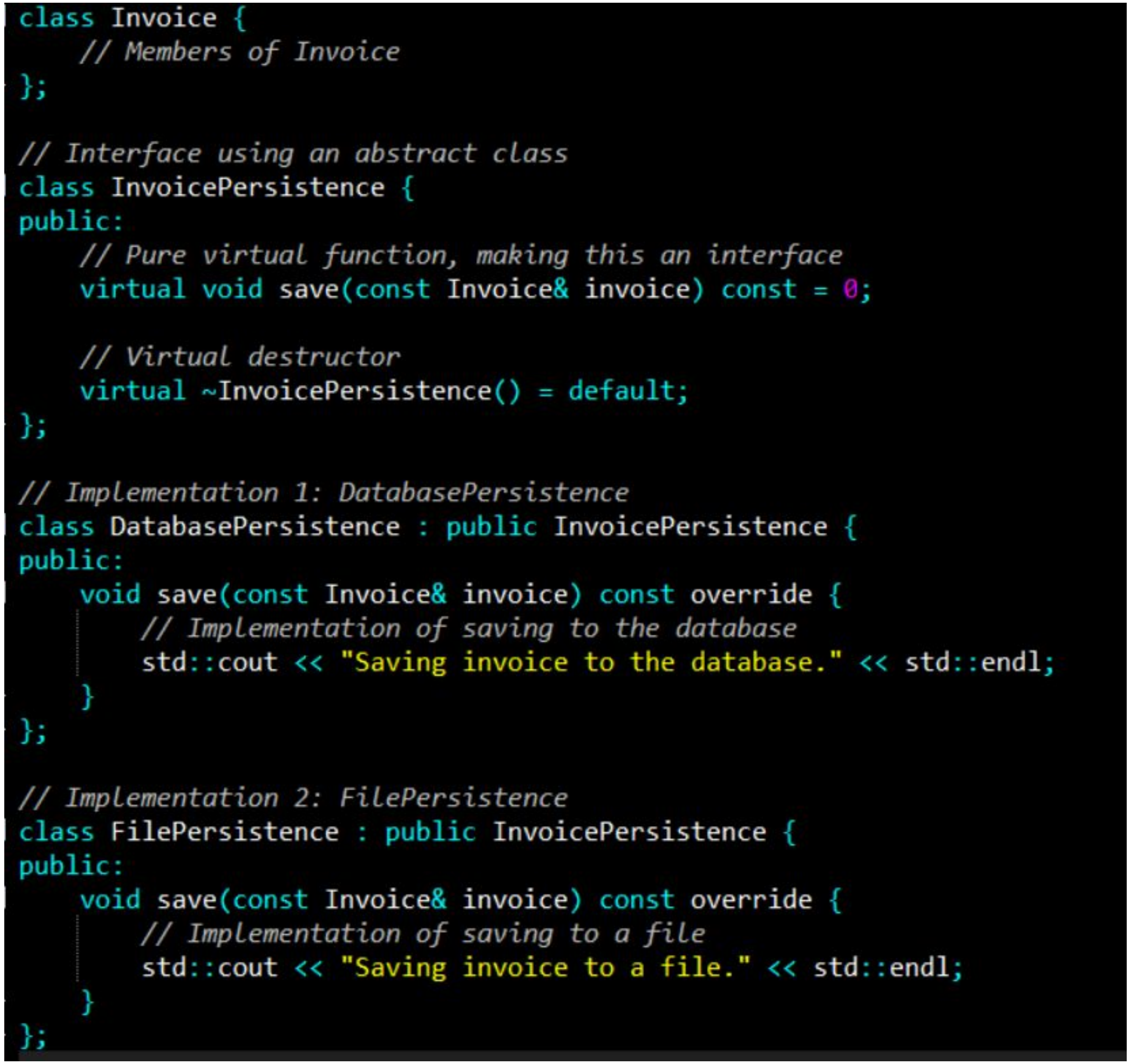
How to Fix?

```
public class FilePersistence implements InvoicePersistence {
    // Override
    public void save(Invoice invoice) {
        // Save to file
    }
}
```

OCP-Open Closed Principle

Our class structure now looks like this:





```cpp
class Invoice {
    // Members of Invoice
};

// Interface using an abstract class
class InvoicePersistence {
public:
    // Pure virtual function, making this an interface
    virtual void save(const Invoice& invoice) const = 0;

    // Virtual destructor
    virtual ~InvoicePersistence() = default;
};

// Implementation 1: DatabasePersistence
class DatabasePersistence : public InvoicePersistence {
public:
    void save(const Invoice& invoice) const override {
        // Implementation of saving to the database
        std::cout << "Saving invoice to the database." << std::endl;
    }
};

// Implementation 2: FilePersistence
class FilePersistence : public InvoicePersistence {
public:
    void save(const Invoice& invoice) const override {
        // Implementation of saving to a file
        std::cout << "Saving invoice to a file." << std::endl;
    }
};
```

OCP-Open Closed Principle

- Now our persistence logic is easily extendable.
- If our boss asks us to add another database and have 2 different types of databases like MySQL and MongoDB, we can easily do that.

Example-2

- Imagine a program that has two instances of a square and needs a custom component to compare their area. The code would look like the following:

```
class Square() {
  int height;
  int area() { return height * height; }
}

public class Example {
  public int compareArea(Square a, Square b) {
    return a.area() - b.area();
  }
}
```

- For this specific use case, the Example class works perfectly well.
- It returns zero if the two squares are the same size, a positive number if the first square is larger and a negative number if the first square is smaller.
- However, a problem quickly arises when a circle is brought into the mix.

Extension Problem

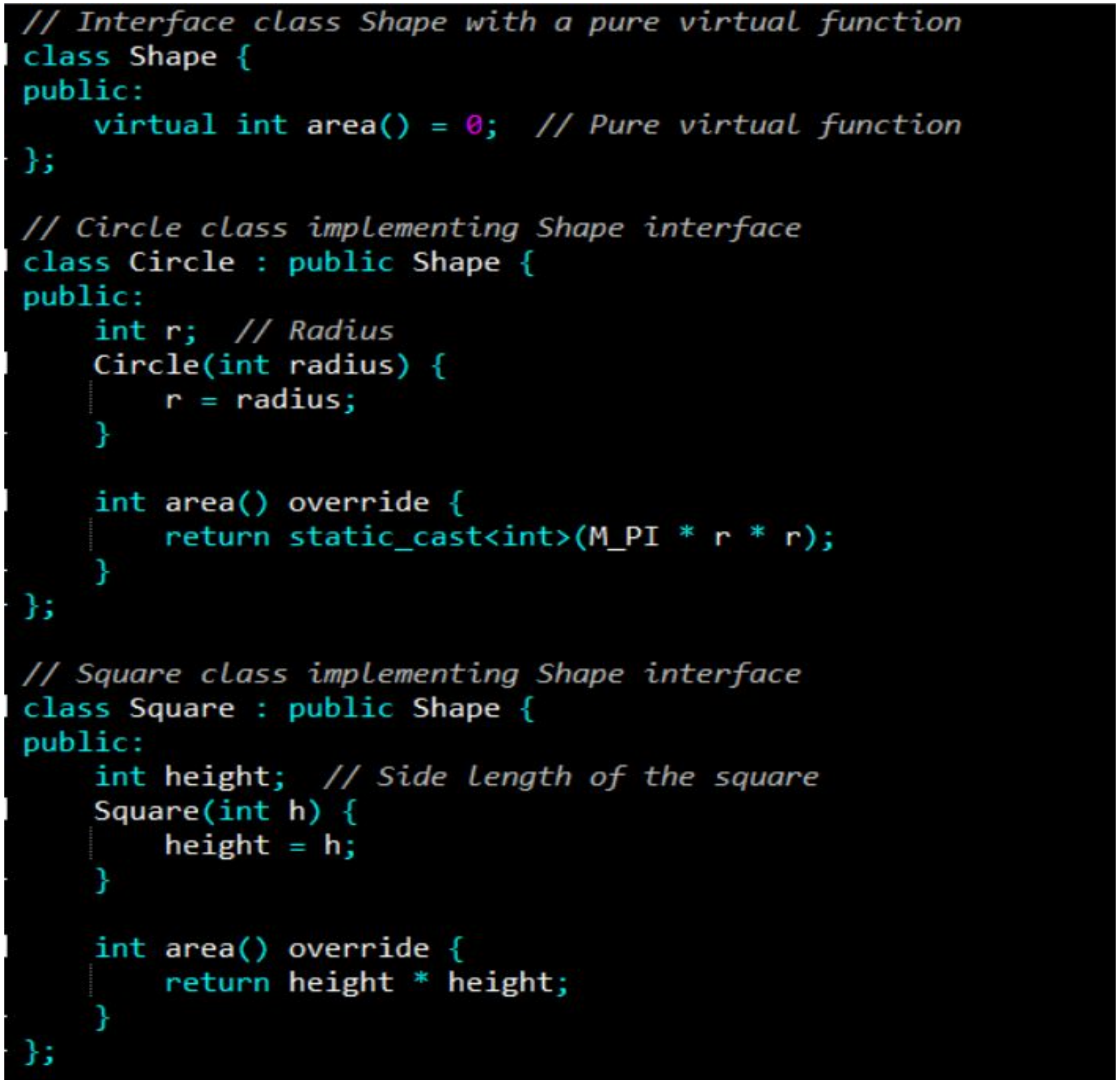
```
class Circle {
  int r;
  int area() { return Math.PI*r*r*;}
}

class Example {

  public int compareArea(Square a, Square b) {
    return a.area() - b.area();
  }
  public int compareArea(Circle x, Circle y) {
   return x.area() - y.area();
  }

}
```

- You can easily imagine the Example class growing larger and larger as more shapes are introduced into the problem domain.
- The addition of an interface to our example helps to overcome the violation of the open-closed principle.
- An interface allows for infinite future extensions with no need to ever edit the class again.
- To fix this example, we first create an interface that both the circle and the square implement

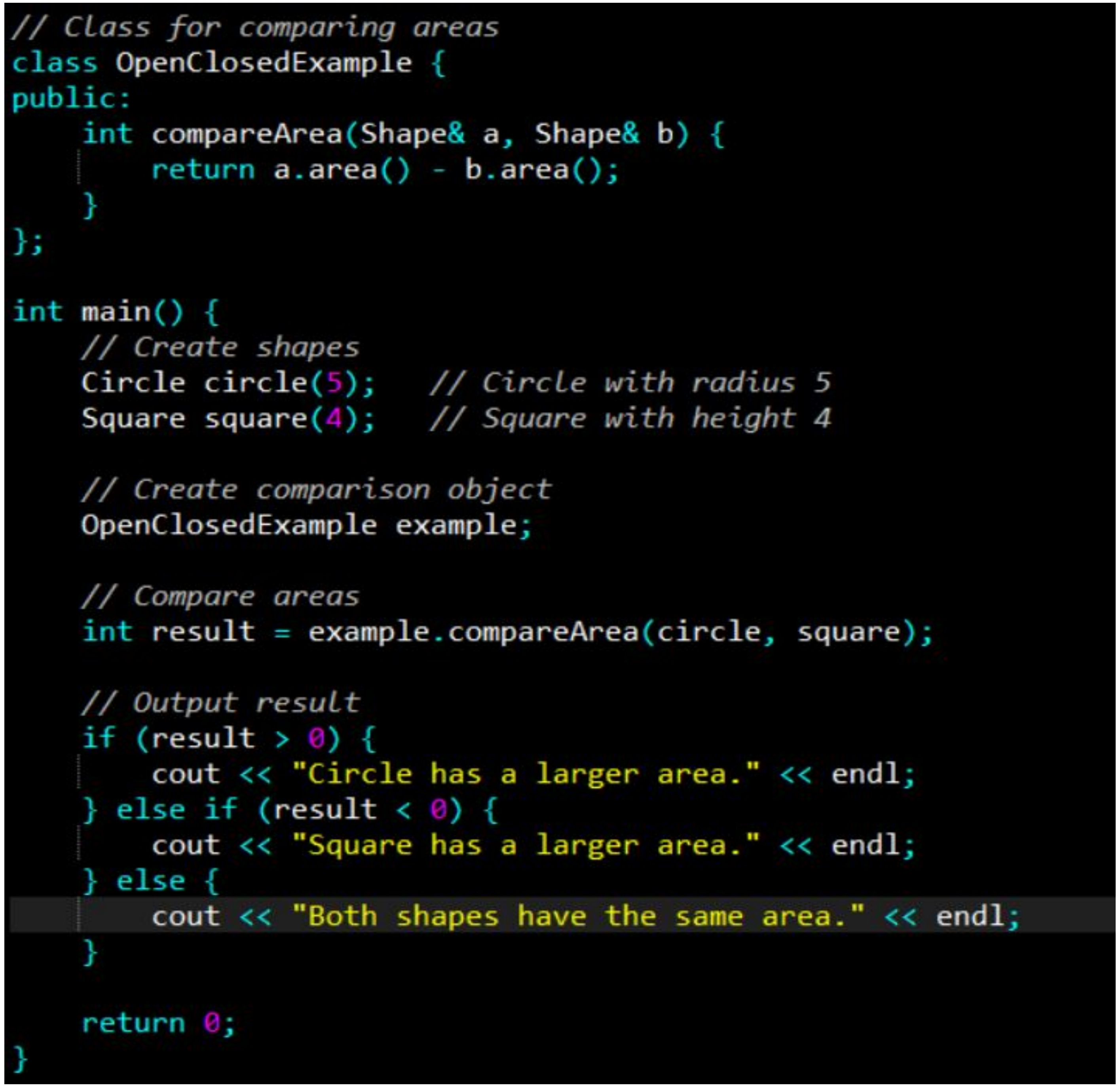
```
interface Shape {
  int area();
}

class Circle implements Shape {
  int r;
  int area() { return Math.PI*r*r*;}
}

class Square implements Shape {
  int height;
  int area() { return height * height; }
}
```

- We then create a new class named OpenClosedExample which has a single compareArea method that uses the Shape interface as arguments:

```
public class OpenClosedExample {


 public int compareArea(Shape a, Shape b) {
  return a.area() - b.area();
 }


}
```



```cpp
// Interface class Shape with a pure virtual function
class Shape {
public:
    virtual int area() = 0;  // Pure virtual function
};

// Circle class implementing Shape interface
class Circle : public Shape {
public:
    int r;  // Radius
    Circle(int radius) {
        r = radius;
    }

    int area() override {
        return static_cast<int>(M_PI * r * r);
    }
};

// Square class implementing Shape interface
class Square : public Shape {
public:
    int height;  // Side length of the square
    Square(int h) {
        height = h;
    }

    int area() override {
        return height * height;
    }
};
```



```
// Class for comparing areas
class OpenClosedExample {
public:
    int compareArea(Shape& a, Shape& b) {
        return a.area() - b.area();
    }
};

int main() {
    // Create shapes
    Circle circle(5);   // Circle with radius 5
    Square square(4);   // Square with height 4

    // Create comparison object
    OpenClosedExample example;

    // Compare areas
    int result = example.compareArea(circle, square);

    // Output result
    if (result > 0) {
        cout << "Circle has a larger area." << endl;
    } else if (result < 0) {
        cout << "Square has a larger area." << endl;
    } else {
        cout << "Both shapes have the same area." << endl;
    }

    return 0;
}
```

jo class problem create kar rhi he us ki jaga aik new class bana do us problem ki or usko attach kardo main class ke sath or jo new he us se inherit karwa do wo old class jiski jaga new class banayi thi

LSP-Liskov Substitution Principle

- The Liskov Substitution Principle states that **subclasses should be substitutable for their base classes**.
- This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case.
- This is the expected behavior, because when we use inheritance, we assume that the child class inherits everything that the superclass has. The child **class extends the behavior but never narrows it down**.
- Therefore, when a class does not obey this principle, it leads to some bugs that are hard to detect.

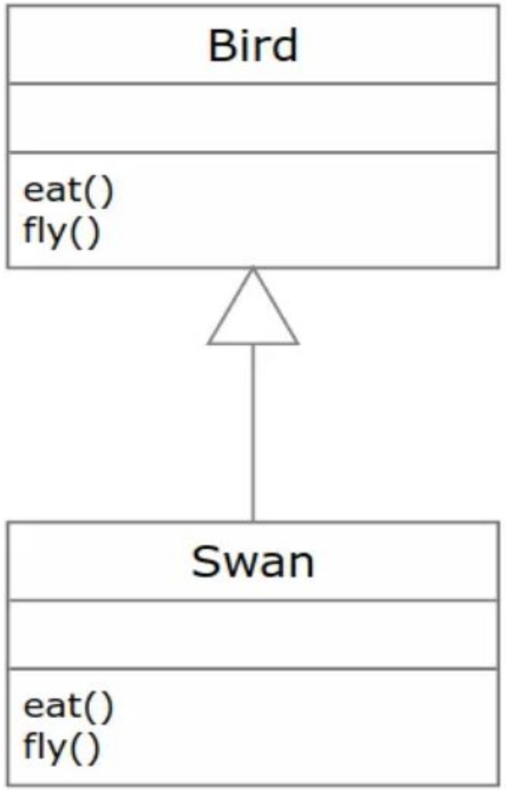
Example (Bird)

```
// Base class Bird
class Bird {
public:
    virtual void fly() {
        cout << "I can fly!" << endl;
    }
    virtual void eat() {
        cout << "I can eat!" << endl;
    }
};
```

```
class Bird {
public:
    virtual void fly() {}
};

class Sparrow : public Bird {
public:
    void fly() override {}
};

class Ostrich : public Bird {
public:
    void fly() override { throw std::runtime_error("Can't fly"); } // Violates LSP
};

// Correct approach using LSP
class Bird {
public:
    virtual void move() {}
};

class Sparrow : public Bird {
public:
    void move() override { /* fly */ }
};

class Ostrich : public Bird {
public:
    void move() override { /* run */ }
};
```

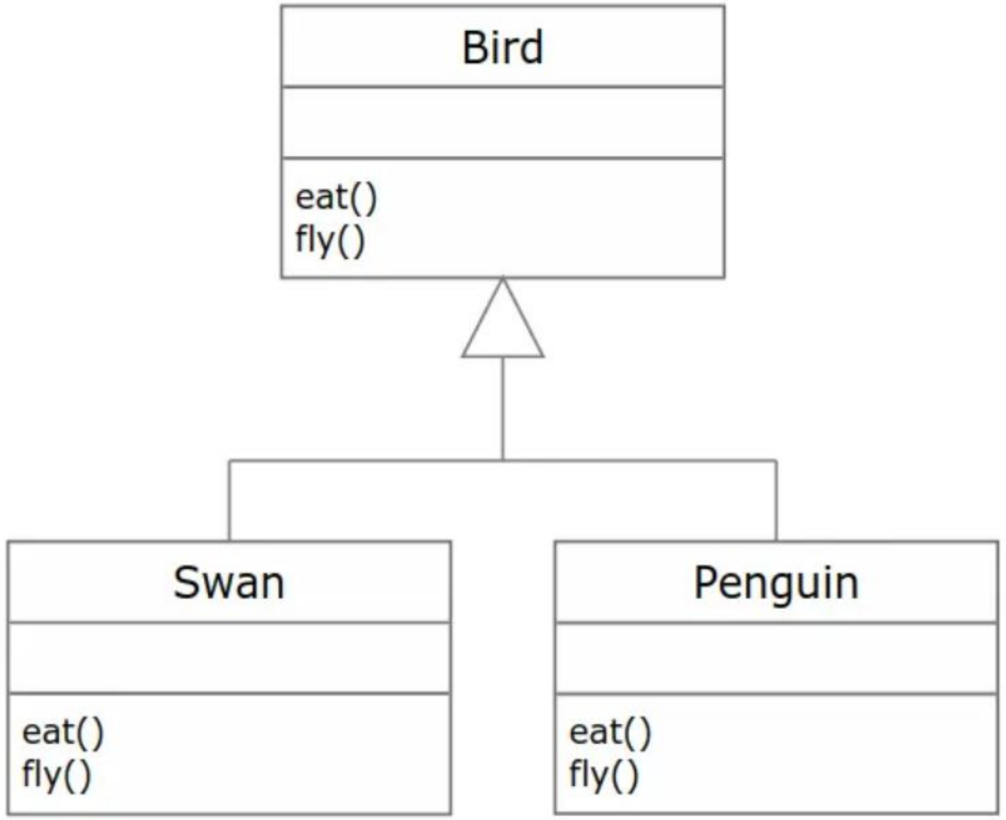
Example (Bird)

```
// Derived class Swan
class Swan : public Bird {
public:
    void fly() override {
        cout << "I believe I can fly!" <<
    }

    void eat() override {
        cout << "I can eat fish and plant
    }
};
```



Example (Bird)

```
// Derived class Penguin
class Penguin : public Bird {
public:
    void fly() override {
        // Penguins cannot fly, this
        throw runtime_error("I cannot
    }

    void eat() override {
        cout << "I can eat fish!" <<
    }
};
```



Example (Bird)

```
// Function to let birds fly
void letBirdsFly(const vector<Bird*>& birds) {
    for (const auto& bird : birds) {
        bird->fly(); // Call the fly method on each bird
    }
}
```

Example (Bird)

```
int main() {
    vector<Bird*> birds; // Create a vector of Bird pointers
    birds.push_back(new Swan());
    birds.push_back(new Penguin());

    // This will throw an exception due to the Penguin
    try {
        letBirdsFly(birds);
    } catch (const std::exception& e) {
        cout << "Error: " << e.what() << endl;
    }
}
```

How Code Violates LSP

- Base Class Expectation: The Bird class defines a fly() method that implies that all derived classes (like Swan and Penguin) should be able to "fly." The expectation is that any instance of a Bird can perform the actions defined in the base class without issue.
- Penguin Implementation: The Penguin class overrides the fly() method to throw an exception:

```
void fly() override {
        // Penguins cannot fly, this is not a valid behavior
        throw std::runtime_error("I cannot fly!");
    }
```

How Code Violates LSP

- Penguins do not meet the expectation set by the Bird class. When the program attempts to invoke fly() on a Penguin object, it fails by throwing an exception, which breaks the expected behavior of the system.
- This exception disrupts the flow of the program and leads to a crash unless handled with a try-catch block.

Example (Bird) Fix

```
// Function to let birds fly, skipping Penguins
void letBirdsFly(const vector<Bird*>& birds) {
    for (const auto& bird : birds) {
        // Attempt to cast to Penguin. If nullptr, it's not a Penguin
        if (dynamic_cast<Penguin*>(bird) == nullptr) {
            bird->fly();
        }
    }
}
```

- But this solution is considered a bad practice, and it **violates the Open-Closed Principle.**
- Imagine if we add another three types of birds that cannot fly. The code is going to become a mess.

Example (Bird) Fix

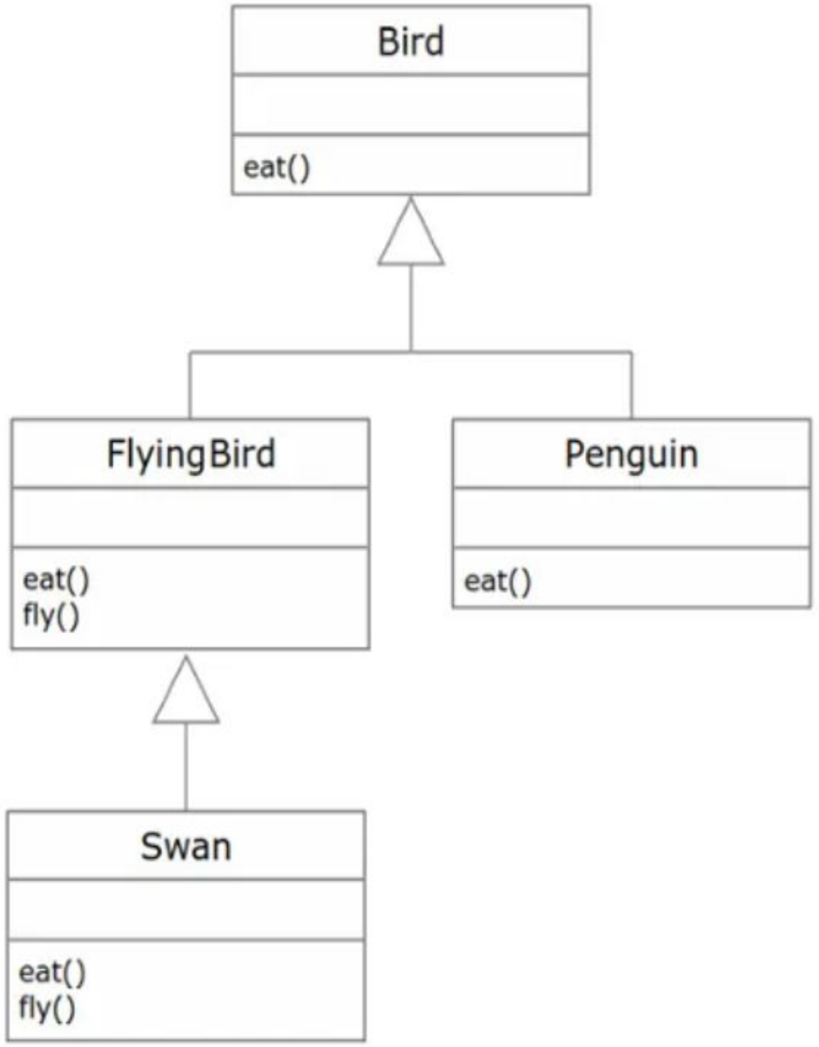
```
// Base class Bird
class Bird {
public:
    virtual void eat() {
        cout << "I can eat!" << endl;
    }
};

// Interface for Flying Birds
class FlyingBird : public Bird {
public:
    virtual void fly() = 0;
};
```

Example (Bird) Fix

```
// Derived class Swan
class Swan : public FlyingBird {
public:
    void fly() override {
        cout << "I believe I can fly!" << endl;
    }
    void eat() override {
        cout << "I can eat fish and plants!" << endl;
    }
};
```

Example (Bird) Fix

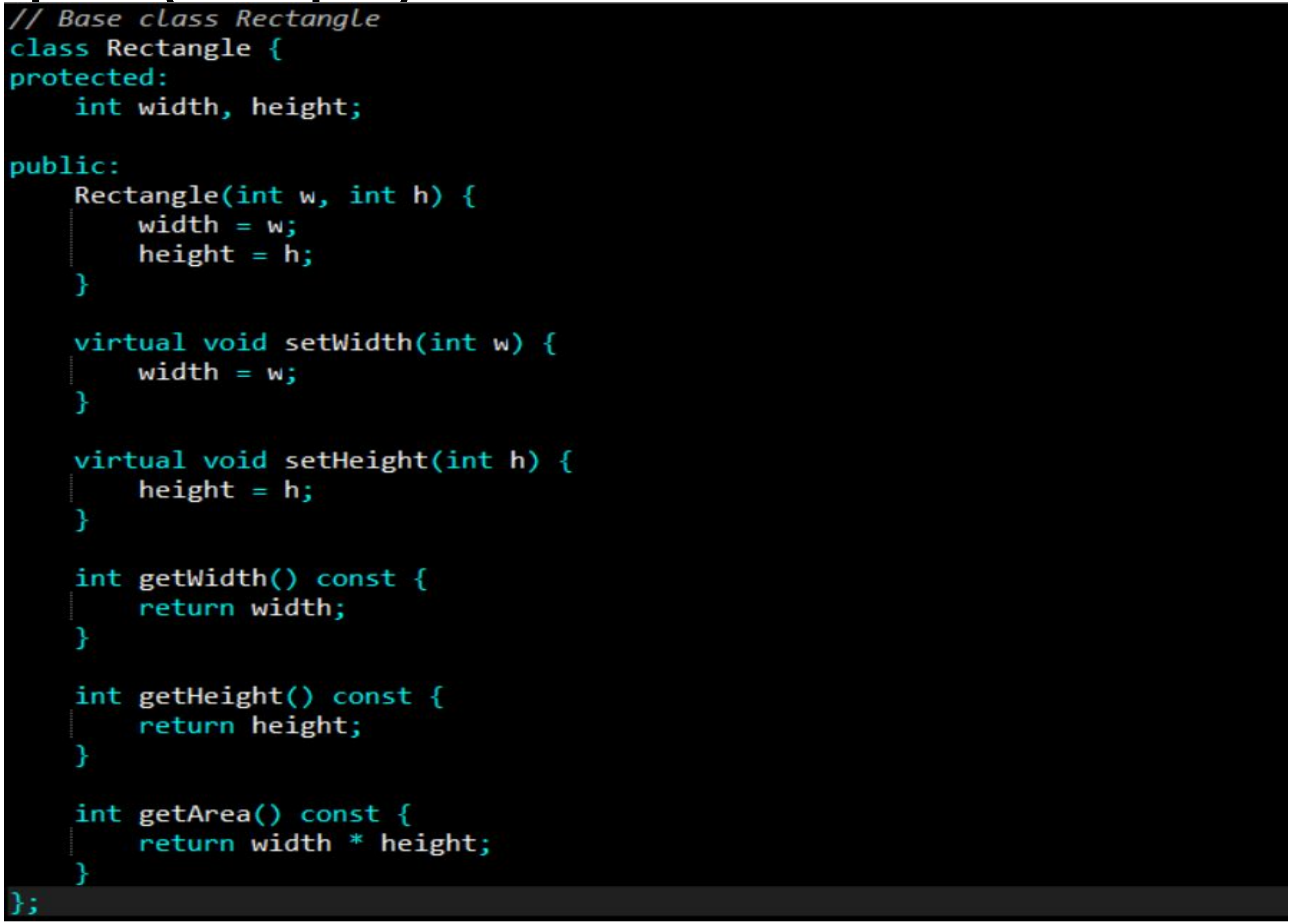
```
// Derived class Penguin
class Penguin : public Bird {
public:
    void eat() override {
        cout << "I can eat fish!" << endl;
    }
};
// Function to let flying birds fly
void letFlyingBirdsFly(const vector<FlyingBird*>& birds) {
    for (const auto& bird : birds) {
        bird->fly();
    }
}
```



Example (Bird) Fix

```
int main() {
    vector<FlyingBird*> flyingBirds;
    flyingBirds.push_back(new Swan());

    // Let all flying birds fly
    letFlyingBirdsFly(flyingBirds);

    // Create a Penguin (does not need to fly)
    Penguin penguin;
    penguin.eat(); // This is fine
}
```

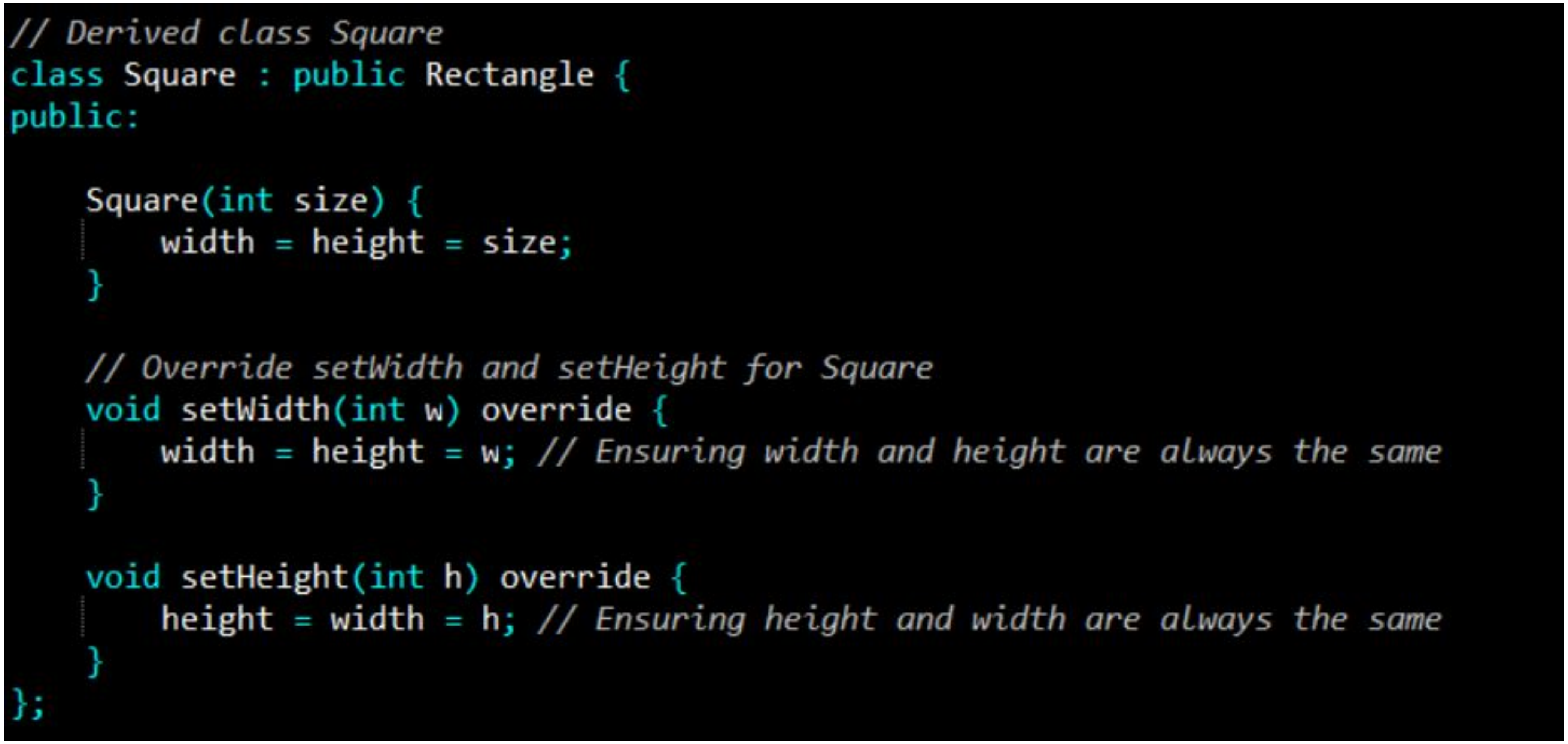
Example (Shape)



```
// Base class Rectangle
class Rectangle {
protected:
    int width, height;

public:
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    virtual void setWidth(int w) {
        width = w;
    }

    virtual void setHeight(int h) {
        height = h;
    }

    int getWidth() const {
        return width;
    }

    int getHeight() const {
        return height;
    }

    int getArea() const {
        return width * height;
    }
};
```

- Now we decide to create another class for Squares.
- As you might know, a square is just a special type of rectangle where the width is equal to the height.

Example (Shape)

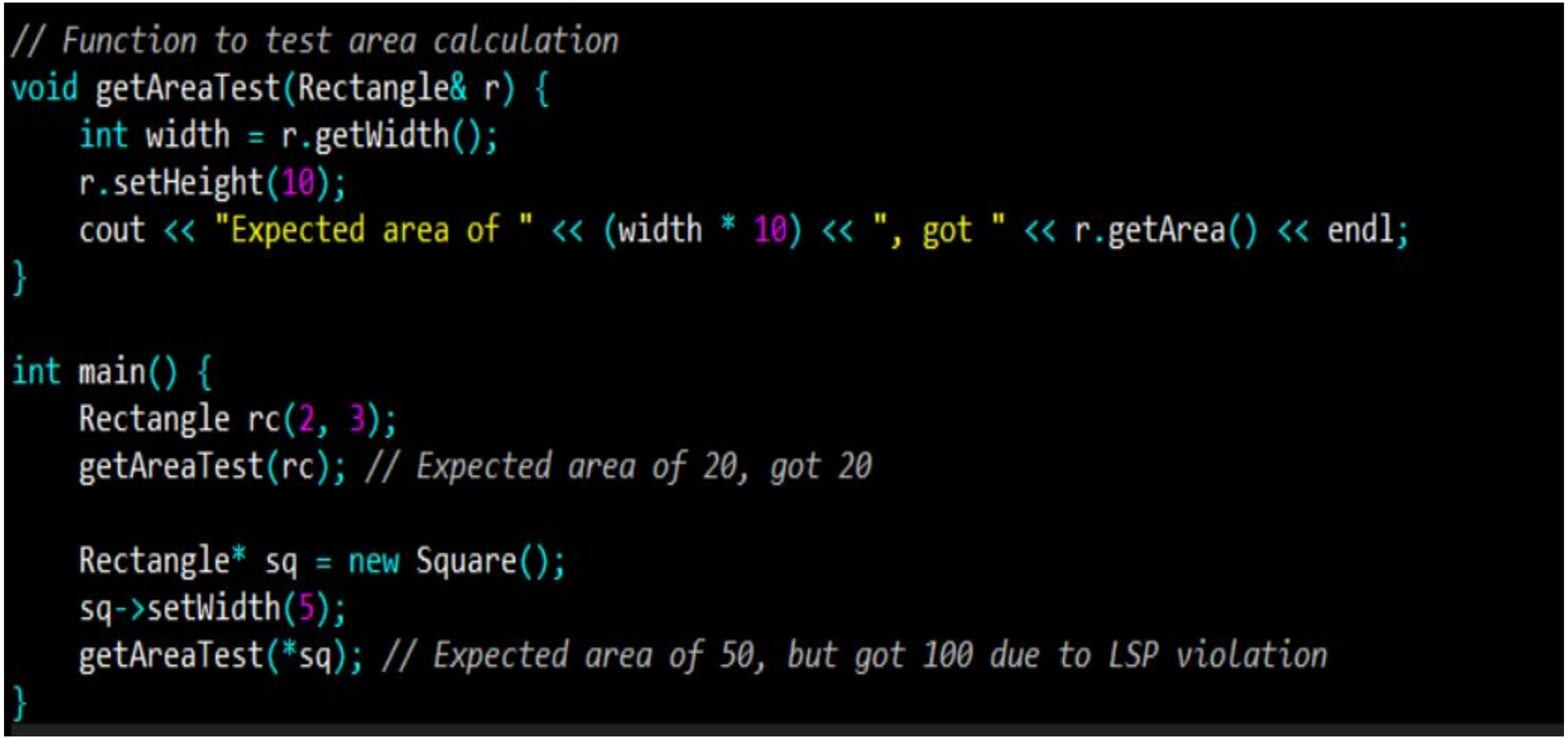


```
// Derived class Square
class Square : public Rectangle {
public:

    Square(int size) {
        width = height = size;
    }

    // Override setWidth and setHeight for Square
    void setWidth(int w) override {
        width = height = w; // Ensuring width and height are always the same
    }

    void setHeight(int h) override {
        height = width = h; // Ensuring height and width are always the same
    }
};
```

Violation

- Our Square class extends the Rectangle class. We set height and width to the same value in the constructor, but we do not want any client (someone who uses our class in their code) to change height or width in a way that can violate the square property.
- Therefore, we override the setters to set both properties whenever one of them is changed. But by doing that we have just violated the Liskov substitution principle.

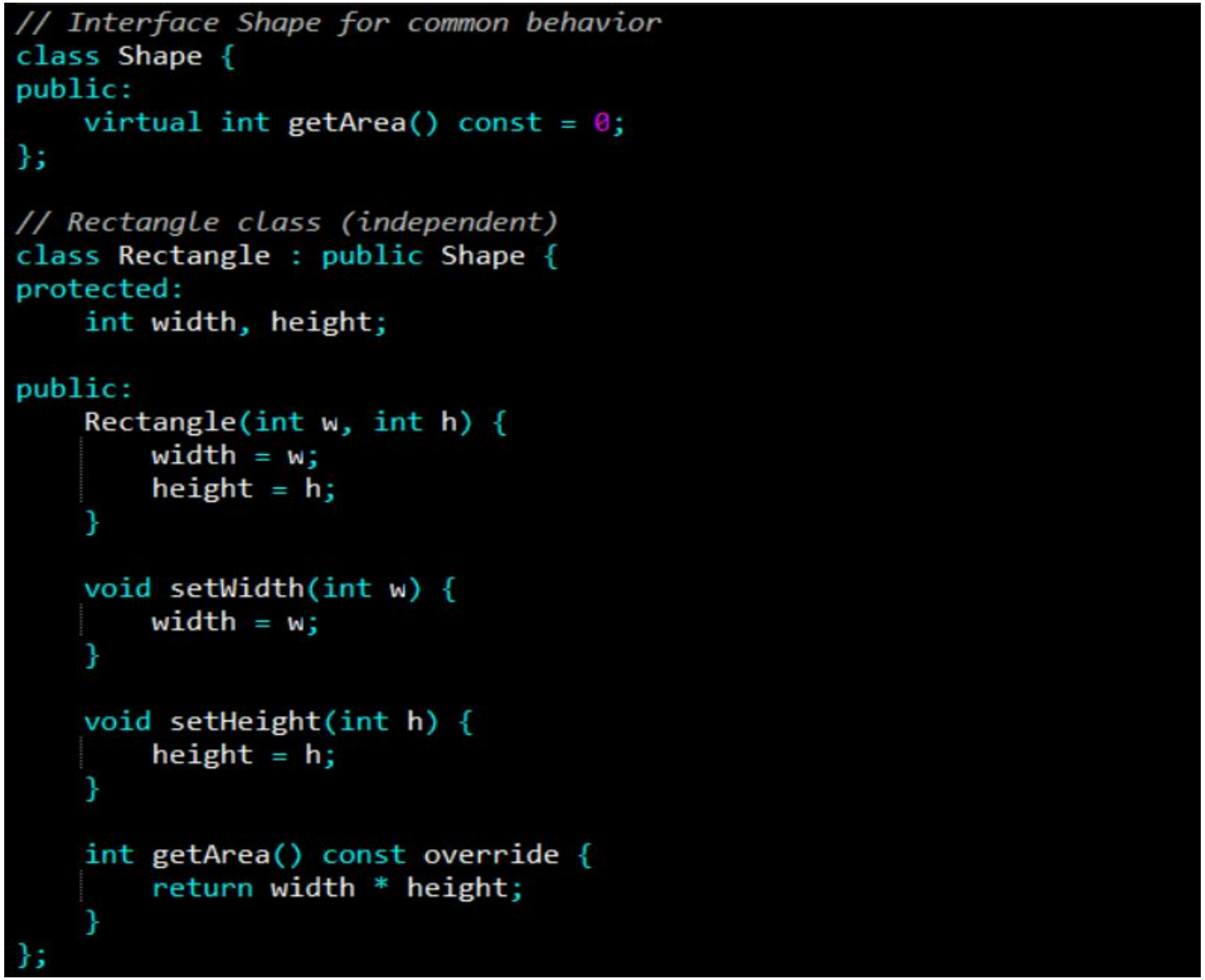
Example (Shape)



```
// Function to test area calculation
void getAreaTest(Rectangle& r) {
    int width = r.getWidth();
    r.setHeight(10);
    cout << "Expected area of " << (width * 10) << ", got " << r.getArea() << endl;
}

int main() {
    Rectangle rc(2, 3);
    getAreaTest(rc); // Expected area of 20, got 20

    Rectangle* sq = new Square();
    sq->setWidth(5);
    getAreaTest(*sq); // Expected area of 50, but got 100 due to LSP violation
}
```

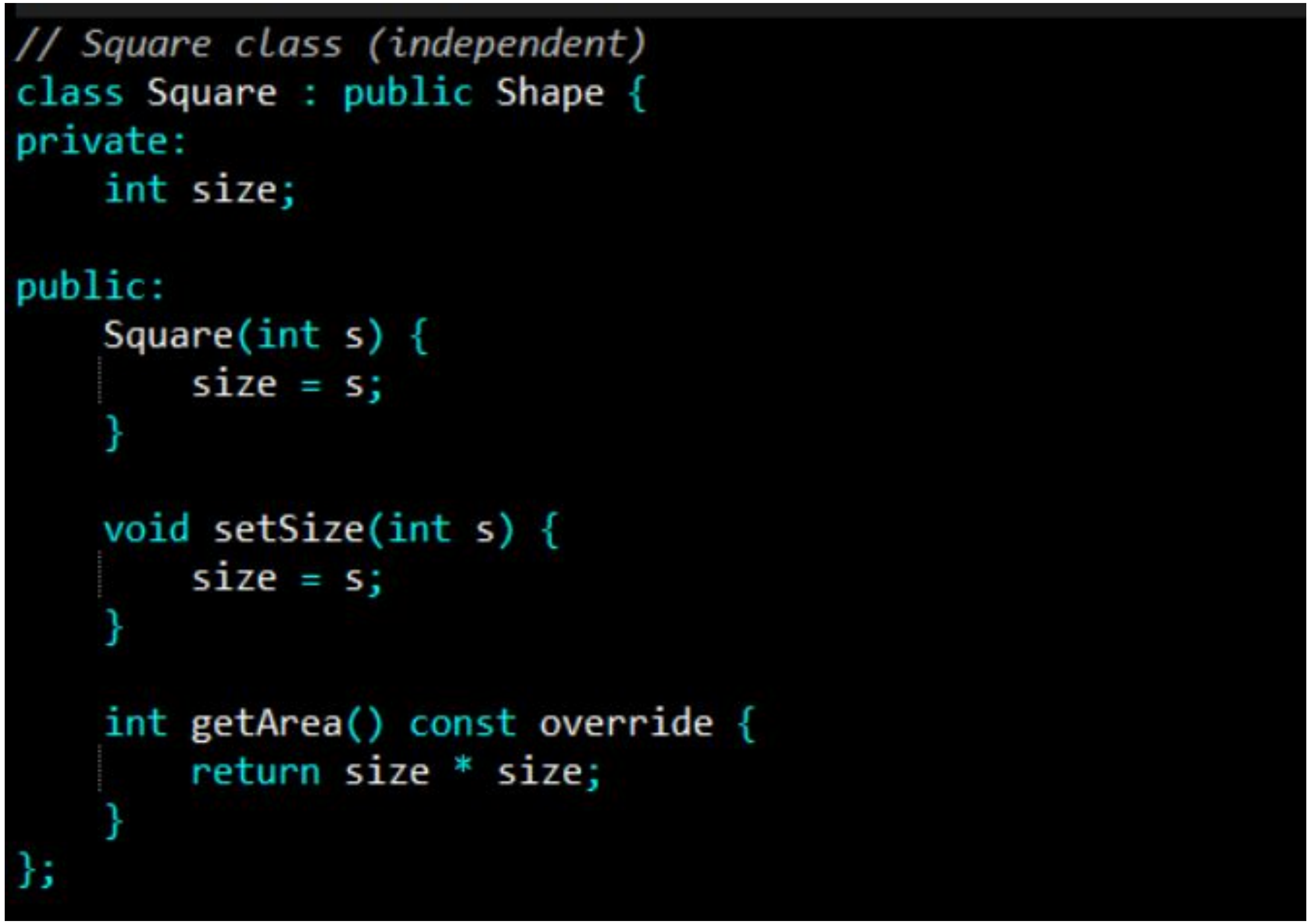
- Your team's tester just came up with the testing function **getAreaTest** and tells you that your **getAreaTest** function fails to pass the test for square objects.
- In the first test, we create a rectangle where the width is 2 and the height is 3 and call **getAreaTest.** The output is 20 as expected, but things go wrong when we pass in the square.
- This is because the call to **setHeight** function in the test is setting the width as well and results in an unexpected output.

Example (Shape) Fix



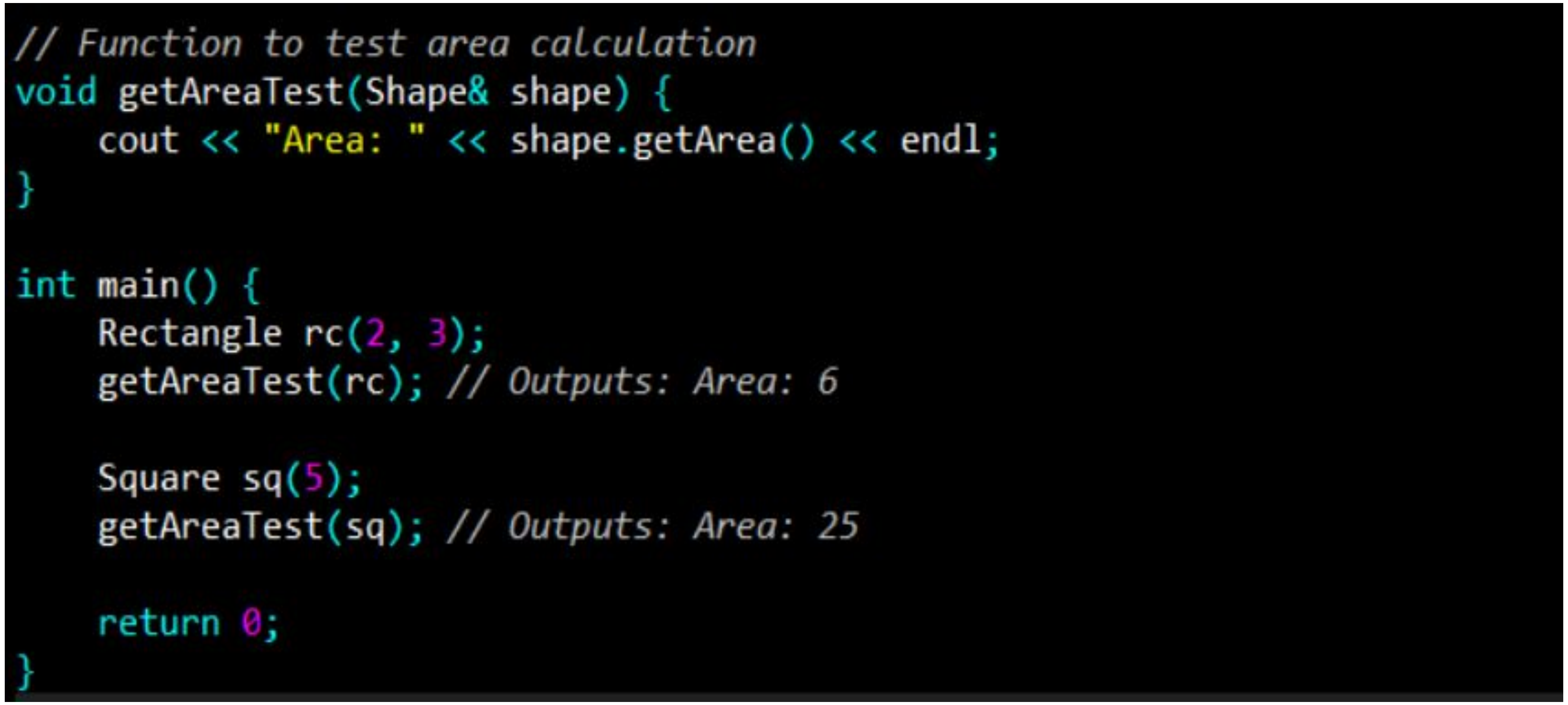
```cpp
// Interface Shape for common behavior
class Shape {
public:
    virtual int getArea() const = 0;
};

// Rectangle class (independent)
class Rectangle : public Shape {
protected:
    int width, height;

public:
    Rectangle(int w, int h) {
        width = w;
        height = h;
    }

    void setWidth(int w) {
        width = w;
    }

    void setHeight(int h) {
        height = h;
    }

    int getArea() const override {
        return width * height;
    }
};
```

Example (Shape) Fix



```cpp
// Square class (independent)
class Square : public Shape {
private:
    int size;

public:
    Square(int s) {
        size = s;
    }

    void setSize(int s) {
        size = s;
    }

    int getArea() const override {
        return size * size;
    }
};
```

Example (Shape) Fix



```cpp
// Function to test area calculation
void getAreaTest(Shape& shape) {
    cout << "Area: " << shape.getArea() << endl;
}

int main() {
    Rectangle rc(2, 3);
    getAreaTest(rc); // Outputs: Area: 6

    Square sq(5);
    getAreaTest(sq); // Outputs: Area: 25

    return 0;
}
```

References

- Clean Architecture: A Craftsman's Guide to Software Structure and Design, 1st Edition, Robert C. Martin, Pearson, 2017.