Refactoring as a Design Activity

Object-oriented Design, Design Principles and Patterns

We need to…

- …structure functionalities as the code scales up so we can continue understand and maintain it
- …find the right boundaries for objects so that they play well with their neighbors

We want…

- …objects to represent coherent units that make sense in its larger environment

- …to build flexible systems

Messages

The big idea is “messaging” [...] The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.

Alan Kay - Email Message Sent to the Squeak Mailing List

Web of Objects

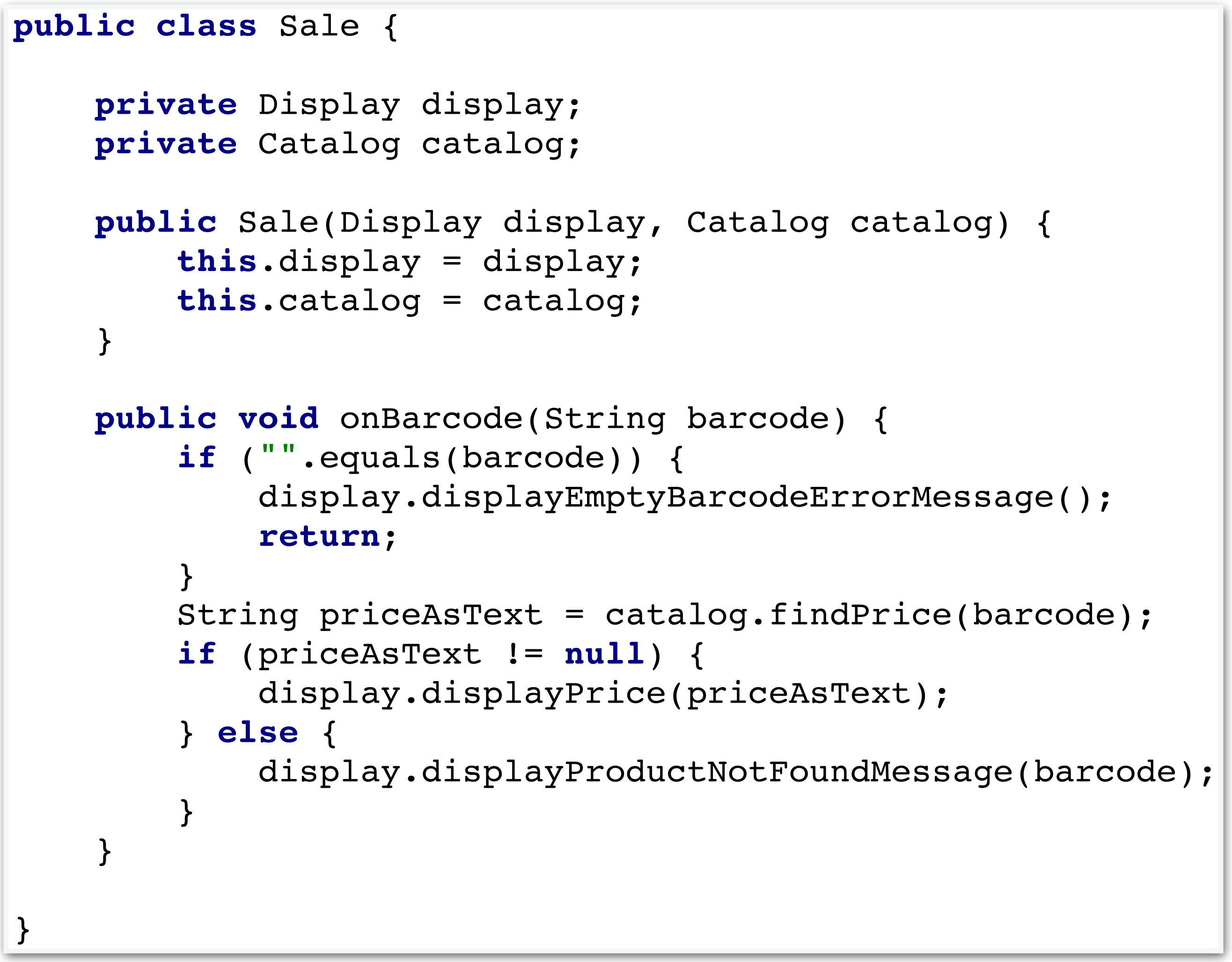
An object-oriented system is built by creating **objects** and plugging them together so that they can send **messages** to one another.

Values

- Immutable instances that model fixed quantities
- No individual identities
- Example: Java strings

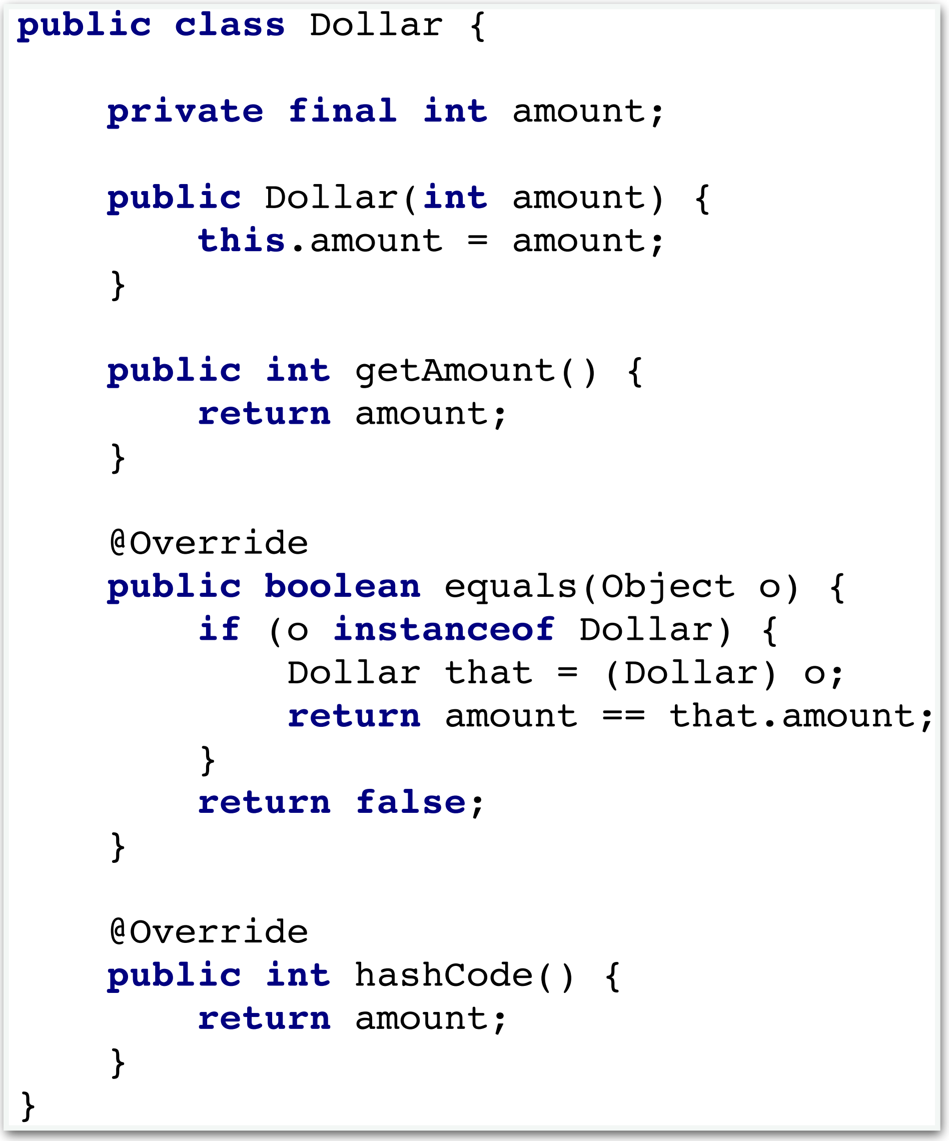
Objects

- Use mutable state to model their behavior over time
- Two objects of the same type have separate identities even if they have the same state



```java
public class Sale {

    private Display display;
    private Catalog catalog;

    public Sale(Display display, Catalog catalog) {
        this.display = display;
        this.catalog = catalog;
    }

    public void onBarcode(String barcode) {
        if ("".equals(barcode)) {
            display.displayEmptyBarcodeErrorMessage();
            return;
        }
        String priceAsText = catalog.findPrice(barcode);
        if (priceAsText != null) {
            display.displayPrice(priceAsText);
        } else {
            display.displayProductNotFoundMessage(barcode);
        }
    }

}
```

Object



```java
public class Dollar {

    private final int amount;

    public Dollar(int amount) {
        this.amount = amount;
    }

    public int getAmount() {
        return amount;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Dollar) {
            Dollar that = (Dollar) o;
            return amount == that.amount;
        }
        return false;
    }

    @Override
    public int hashCode() {
        return amount;
    }
}
```

Value

Tell, don’t ask
(Law of Demeter)

- Calling object describe what it wants in terms of the role that its neighbor plays

- Called object decides how to make that happen

- Avoid navigating to other objects to make things happen

Train Wreck

```
((EditSaveCustomizer) master.getModelisable()
 .getDockablePanel()
  .getCustomizer()
   .getSaveItem().setEnabled(Boolean.FALSE.booleanValue());
```

This fragment was meant to say

```
master.allowSavingOfCustomizations();
```

But sometime ask…

- Occasionally we ask objects about their state when searching or filtering

- We still want to maintain expressiveness and avoid “train wrecks"

Avoid information leaks…

```
public class Train {

  private final List<Carriage> carriages […]
  private final int percentReserveBarrier = 70;

  public void reserveSeats(ReservationRequest request) {
    for (Carriage carriage : carriages) {
      if (carriage.getSeats().getPercentReserved() < percentReserveBarrier) {
        request.reserveSeatsIn(carriage);
        return;
      }
    }
    request.cannotFindSeats();
  }

}
```

…using the right query

```
public class Train {

  private final List<Carriage> carriages […]
  private final int percentReserveBarrier = 70;

  public void reserveSeats(ReservationRequest request) {
    for (Carriage carriage : carriages) {
      if (carriage.hasSeatsAvailableWithin(percentReserveBarrier)) {
        request.reserveSeatsIn(carriage);
        return;
      }
    }
    request.cannotFindSeats();
  }

}
```

S.O.L.I.D. Principles

- Single Responsibility Principle
- Open-closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle

A class should have only one reason to change.

```
public class Rectangle {

    private double width;
    private double height;
    private Graphics graphics;

    // ...

    public double area() {
        return width * height;
    }

    public void draw() {
        // Do something with Graphics
    }

}
```

Single Responsibility Principle

A class should have only one reason to change.

```java
public class Rectangle {

    private double width;
    private double height;
    private Graphics graphics;

    // ...

    public double area() {
        return width * height;
    }

    public void draw() {
        // Do something with Graphics
    }

}
```

Single Responsibility Principle

A class should have only one reason to change.

```java
public class GeometricRectangle {

    private double width;
    private double height;

    public double area() {
        return width * height;
    }

}

public class Rectangle {

    private GeometricRectangle geometricRectangle;
    private Graphics graphics;

    // ...

    public void draw() {
        // Draw geometricRectangle using Graphics
    }

}
```

Open-closed Principle

Software entities should be open for extension, but closed for modification.

```java
public class Shape {
    // ...
}

public class Rectangle extends Shape {
    // ...
}

public class Circle extends Shape {
    // ...
}

public class GraphicEditor {

    public void drawShape(Shape s) {
        if (s instanceof Rectangle) {
            drawRectangle((Rectangle) s);
        } else if (s instanceof Circle) {
            drawCircle((Circle) s);
        }
    }

    public void drawRectangle(Rectangle rectangle) {
        // ...
    }

    public void drawCircle(Circle c) {
        // ...
    }

}
```

Open-closed Principle

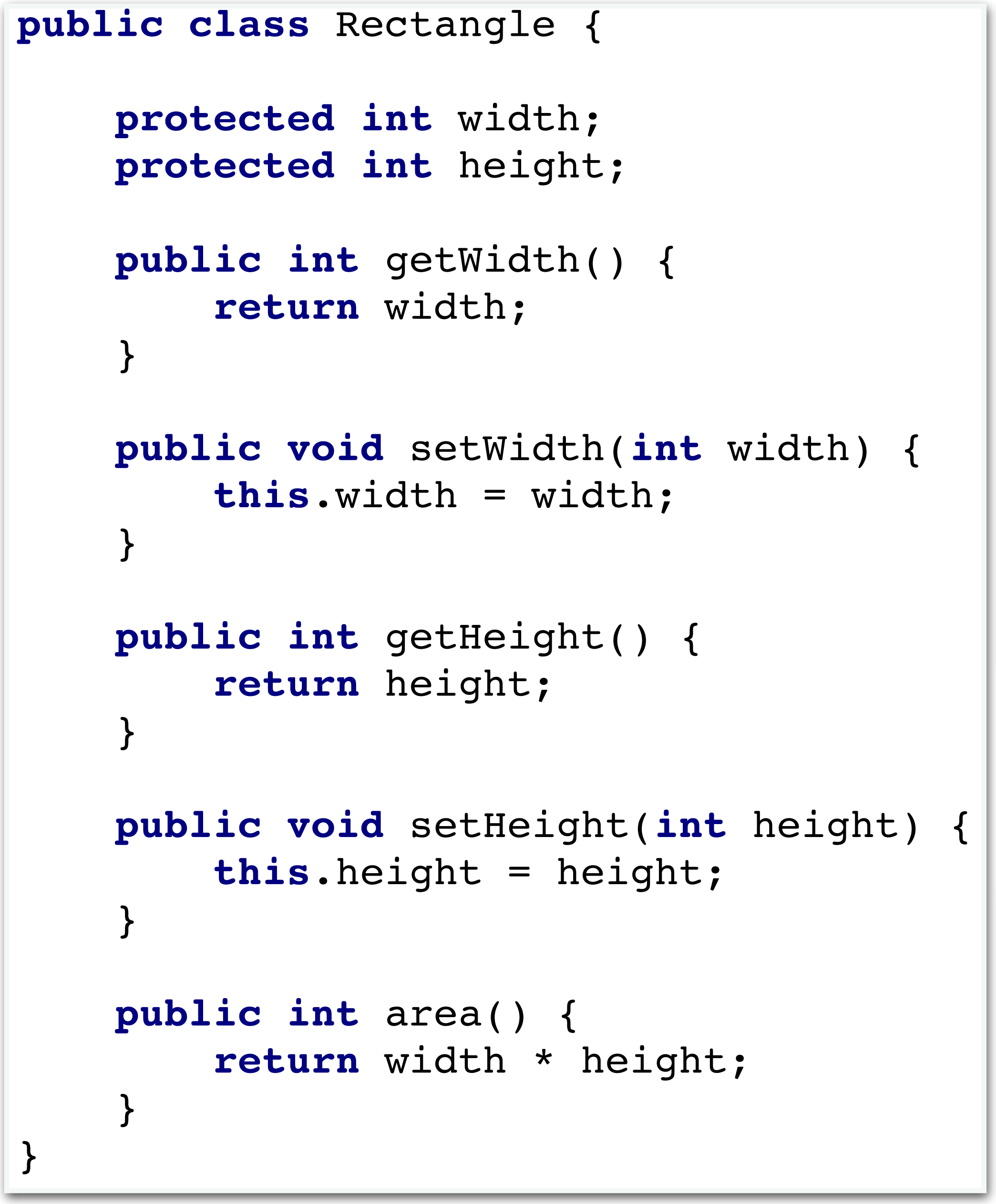
Software entities should be open for extension, but closed for modification.

```java
public class Shape {
    // ...
}

public class Rectangle extends Shape {
    // ...
}

public class Circle extends Shape {
    // ...
}

public class GraphicEditor {

    public void drawShape(Shape s) {
        if (s instanceof Rectangle) {
            drawRectangle((Rectangle) s);
        } else if (s instanceof Circle) {
            drawCircle((Circle) s);
        }
    }

    public void drawRectangle(Rectangle rectangle) {
        // ...
    }

    public void drawCircle(Circle c) {
        // ...
    }

}
```

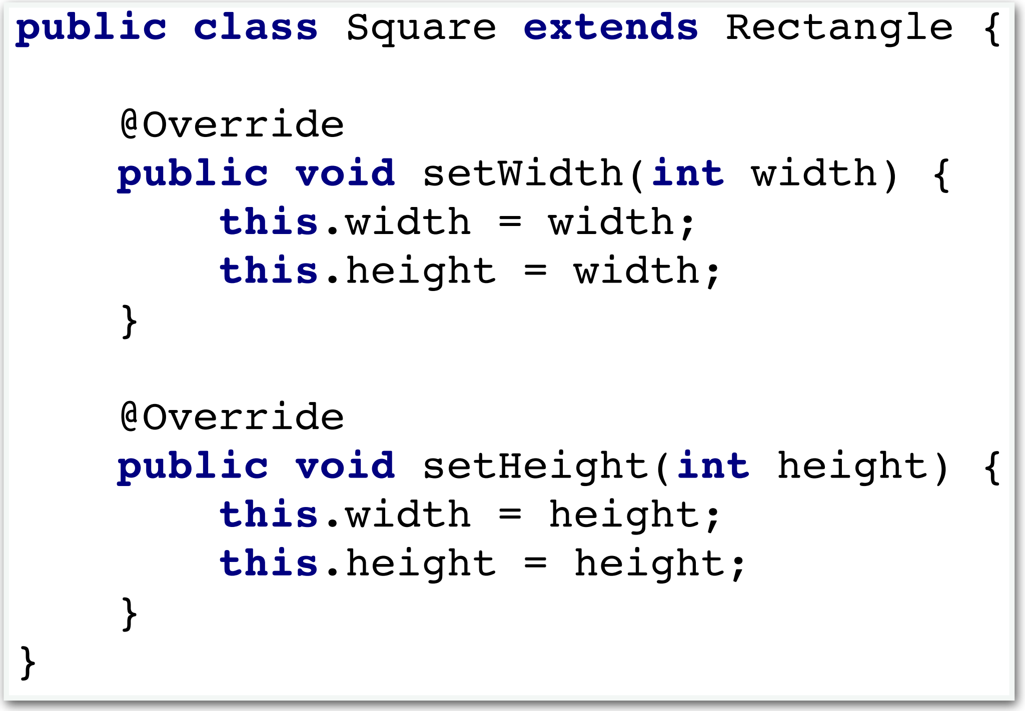
Open-closed Principle

Software entities should be open for extension, but closed for modification.

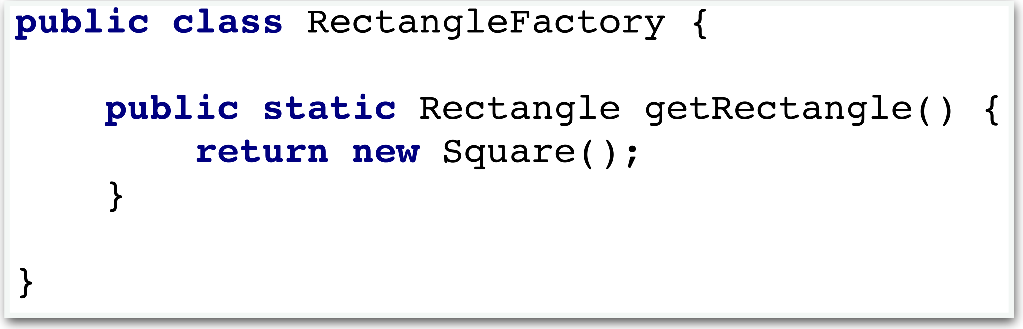
```java
public abstract class Shape {

    // ...

    public abstract void draw();

}

public class Rectangle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the rectangle
    }

}

public class Circle extends Shape {

    // ...

    @Override
    public void draw() {
        // Draw the circle
    }

}

public class GraphicEditor {

    public void drawShape(Shape s) {
        s.draw();
    }

}
```



```java
public class Rectangle {

    protected int width;
    protected int height;

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int area() {
        return width * height;
    }
}
```



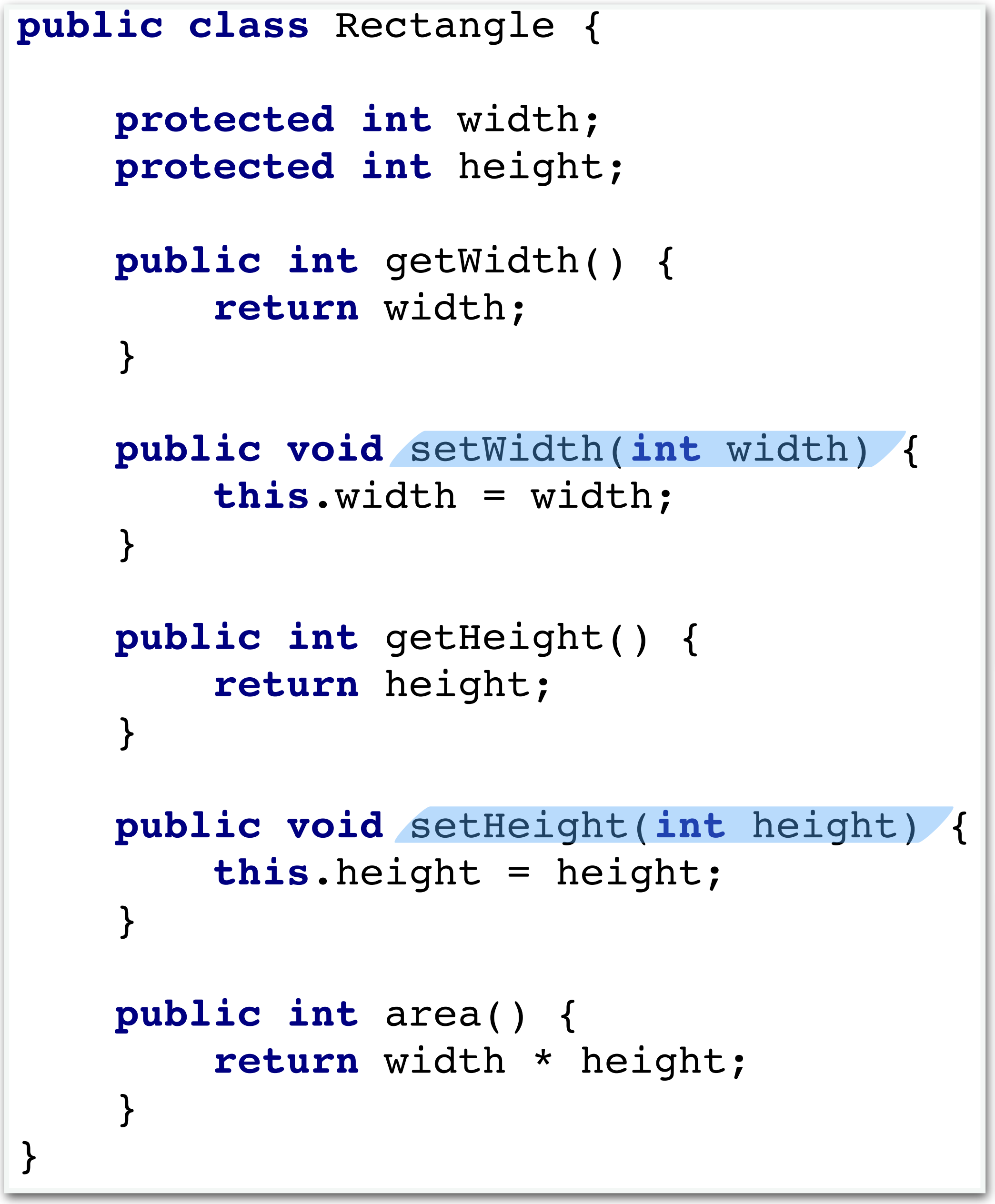
```java
public class Square extends Rectangle {

    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;
    }

    @Override
    public void setHeight(int height) {
        this.width = height;
        this.height = height;
    }
}
```



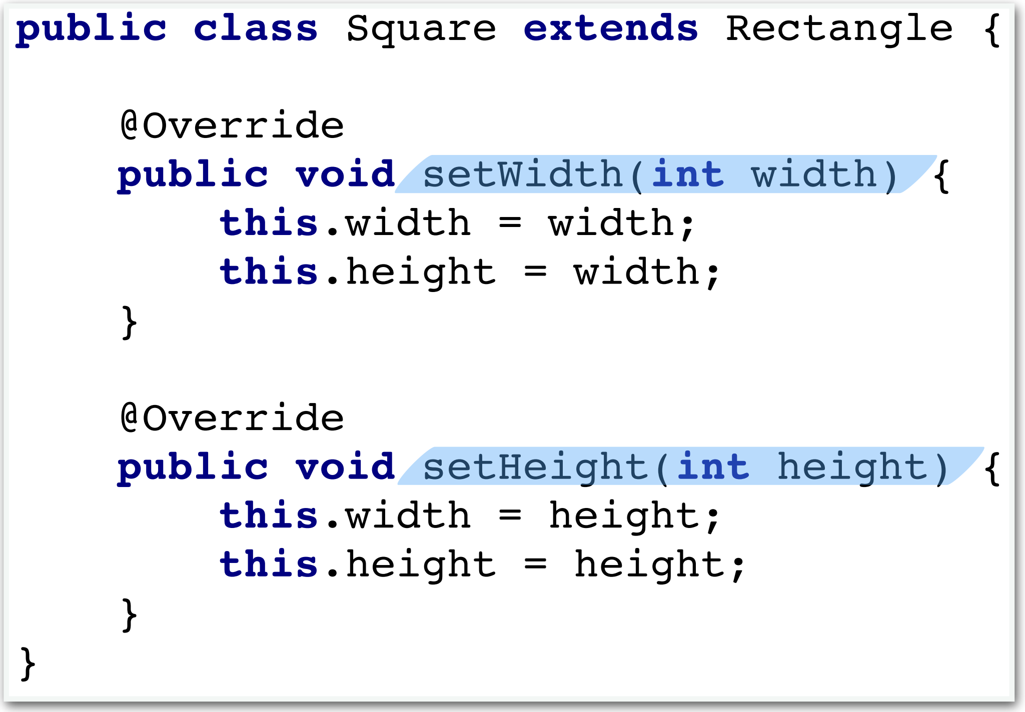
```java
public class RectangleFactory {

    public static Rectangle getRectangle() {
        return new Square();
    }

}
```

Liskov Substitution Principle

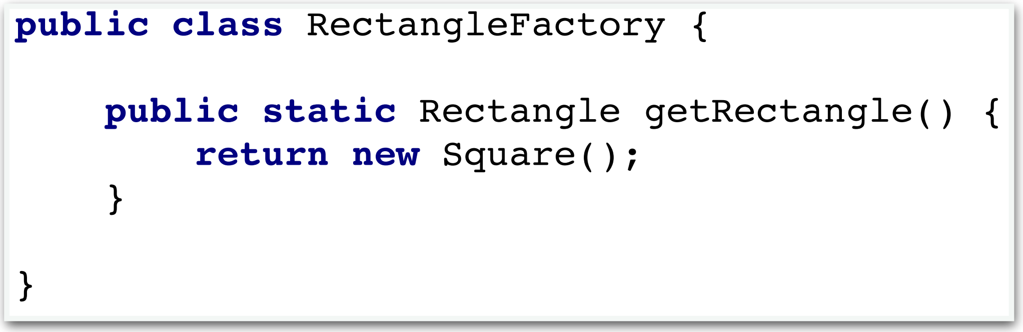
If a method is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the method.



```java
public class Rectangle {

    protected int width;
    protected int height;

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int area() {
        return width * height;
    }
}
```



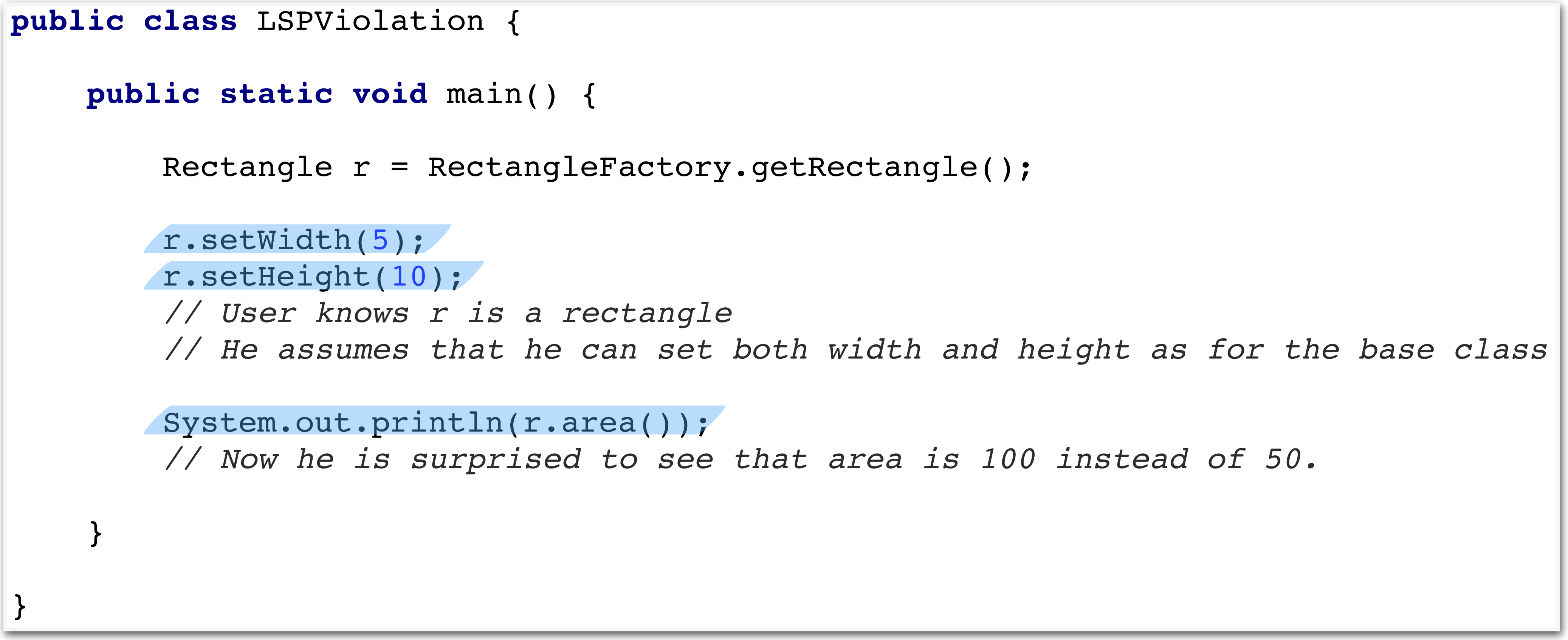
```java
public class Square extends Rectangle {

    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;
    }

    @Override
    public void setHeight(int height) {
        this.width = height;
        this.height = height;
    }
}
```



```java
public class RectangleFactory {

    public static Rectangle getRectangle() {
        return new Square();
    }

}
```

Liskov Substitution Principle

If a method is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the method.



```
public class LSPViolation {

    public static void main() {

        Rectangle r = RectangleFactory.getRectangle();

        r.setWidth(5);
        r.setHeight(10);
        // User knows r is a rectangle
        // He assumes that he can set both width and height as for the base class

        System.out.println(r.area());
        // Now he is surprised to see that area is 100 instead of 50.

    }

}
```

Liskov Substitution Principle

If a method is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the method.

Interface Segregation Principle

Clients should not be forced to depend upon interface members that they don't use.

```java
public interface Worker {

    void work();

    void eat();

}

public class Human implements Worker {

    @Override
    public void work() {
        // ...working
    }

    @Override
    public void eat() {
        // ...eating during break
    }
}

public class SuperHuman implements Worker {

    @Override
    public void work() {
        // ...working much more
    }

    @Override
    public void eat() {
        // ...eating during break
    }
}

public class Manager {

    private Worker worker;

    public void setWorker(Worker worker) {
        this.worker = worker;
    }

    public void manage() {
        worker.work();
    }
}
```

Interface Segregation Principle

Clients should not be forced to depend upon interface members that they don't use.

```java
public interface Worker {

    void work();

    void eat();

}

public class Human implements Worker {

    @Override
    public void work() {
        // ...working
    }

    @Override
    public void eat() {
        // ...eating during break
    }
}

public class SuperHuman implements Worker {

    @Override
    public void work() {
        // ...working much more
    }

    @Override
    public void eat() {
        // ...eating during break
    }
}

public class Manager {

    private Worker worker;

    public void setWorker(Worker worker) {
        this.worker = worker;
    }

    public void manage() {
        worker.work();
    }
}
```

Interface Segregation Principle

Clients should not be forced to depend upon interface members that they don't use.

```java
public interface Worker {

    void work();

}

public interface Eater {

    void eat();

}

public class Human implements Worker, Eater {

    @Override
    public void work() {
        // ...working
    }

    @Override
    public void eat() {
        // ...eating during break
    }
}

public class SuperHuman implements Worker, Eater {

    @Override
    public void work() {
        // ...working much more
    }

    @Override
    public void eat() {
        // ...eating during break
    }
}

public class Robot implements Worker {

    @Override
    public void work() {
        // ...working
    }
}
```

Dependency Inversion Principle

High level classes should not depend on low level classes.

```
public class Human {

    public void work() {
        // ...working
    }

}

public class Manager {

    private Human worker;

    public void setWorker(Human worker) {
        this.worker = worker;
    }

    public void manage() {
        worker.work();
    }
}

public class Robot {

    public void work() {
        // ...working longer
    }

}
```

Dependency Inversion Principle

High level classes should not depend on low level classes.

```java
public class Human {

    public void work() {
        // ...working
    }

}

public class Manager {

    private Human worker;

    public void setWorker(Human worker) {
        this.worker = worker;
    }

    public void manage() {
        worker.work();
    }
}

public class Robot {

    public void work() {
        // ...working longer
    }

}
```

Dependency Inversion Principle

High level classes should not depend on low level classes.

```java
public interface Worker {

    void work();

}

public class Human implements Worker {

    public void work() {
        // ...working
    }

}

public class Robot implements Worker {

    public void work() {
        // ...working much more
    }

}

public class Manager {

    private Worker worker;

    public void setWorker(Worker worker) {
        this.worker = worker;
    }
    public void manage() {
        worker.work();
    }

}
```

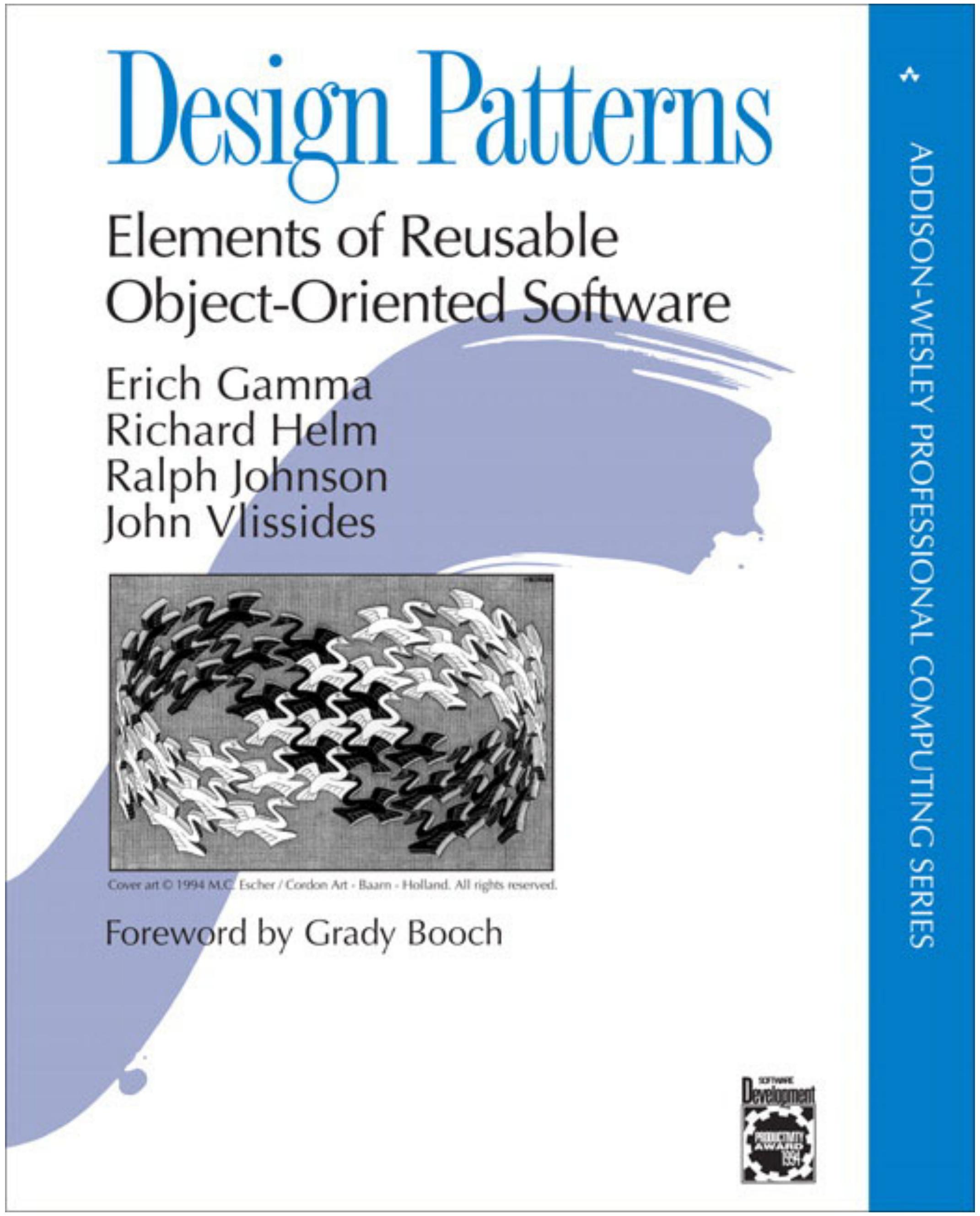
Dependency Inversion Principle

High level classes should not depend on low level classes.

```java
public interface Worker {

    void work();

}

public class Human implements Worker {

    public void work() {
        // ...working
    }

}

public class Robot implements Worker {

    public void work() {
        // ...working much more
    }

}

public class Manager {

    private Worker worker;

    public void setWorker(Worker worker) {
        this.worker = worker;
    }
    public void manage() {
        worker.work();
    }

}
```

Design Patterns

- Important and recurring design in object-oriented systems
- Record experience about how to address problems
- More than one catalog of patterns



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

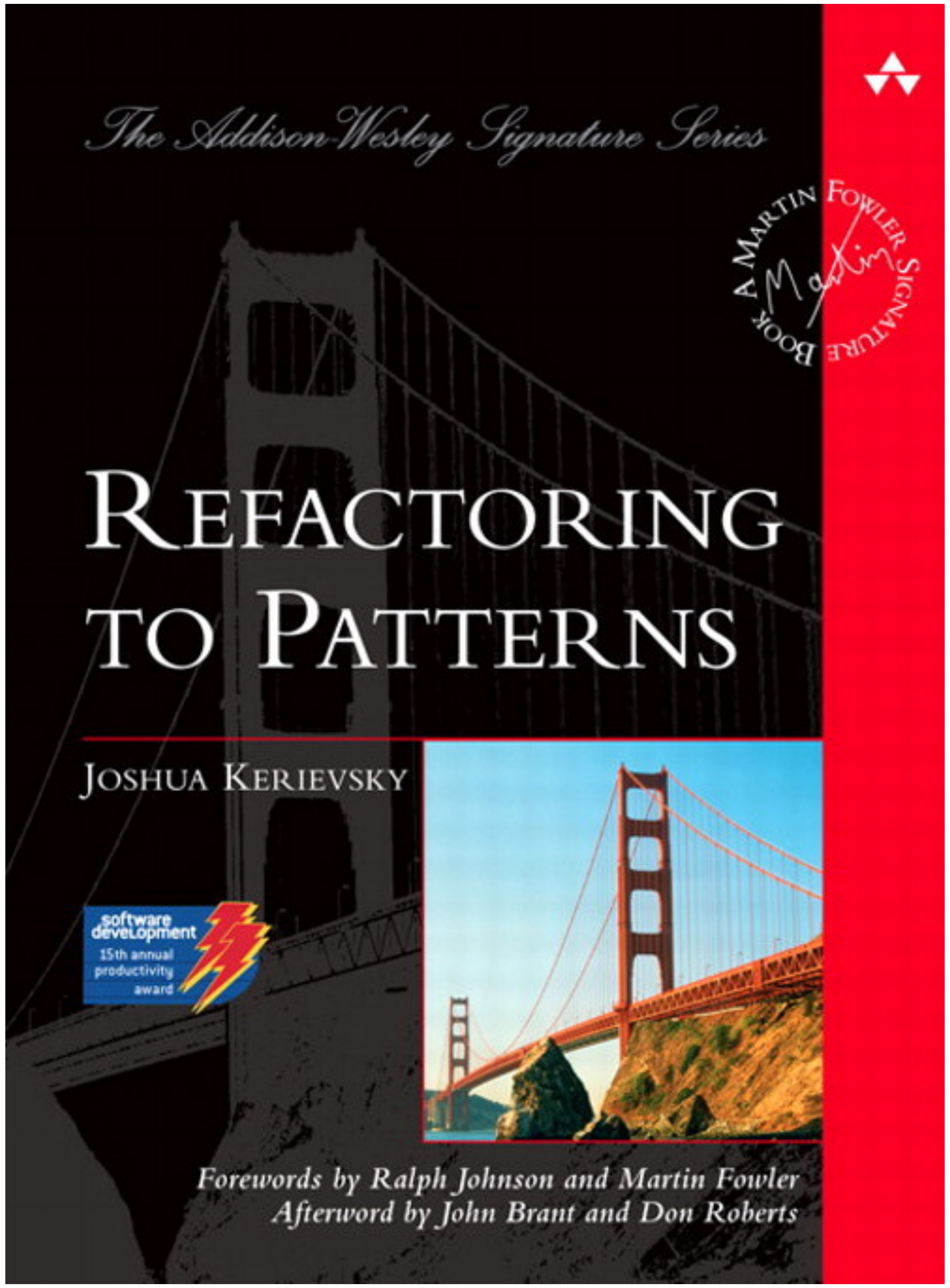
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Refactoring to Patterns

- Refactoring to, towards and away from patterns
- Remove duplication, simplify, communicate intentions
- Deodorize smells


The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Refactoring to Patterns
Joshua Kerievsky
software development 15th annual productivity award
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts