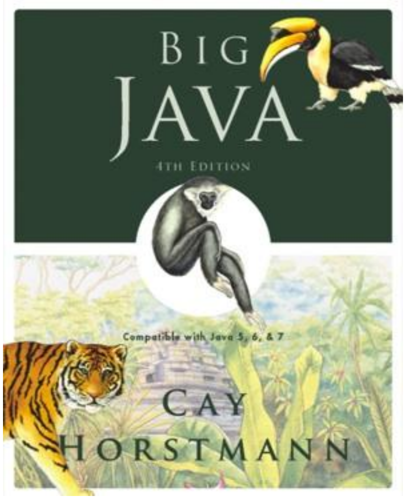

Chapter 8 – Designing Classes

Chapter Goals

- To learn how to discover appropriate classes for a given problem
- To understand the concepts of cohesion and coupling
- To minimize the use of side effects
- To understand the scope rules for local variables and instance variables

Discovering Classes

- A class represents a single concept from the problem domain
- Name for a class should be a noun that describes concept
- Concepts from mathematics:

```
Point
Rectangle
Ellipse
```

- Concepts from real life:

```
BankAccount
CashRegister
```

Cohesion

- A class should represent a single concept
- The public interface of a class is *cohesive* if all of its features are related to the concept that the class represents
- This class lacks cohesion:

```
public class CashRegister
{
   public void enterPayment(int dollars, int quarters,
      int dimes, int nickels, int pennies)
   ...
   public static final double NICKEL_VALUE = 0.05;
   public static final double DIME_VALUE = 0.1;
   public static final double QUARTER_VALUE = 0.25;
   ...
}
```

Cohesion

- `CashRegister`, as described above, involves two concepts: *cash register* and *coin*
- Solution: Make two classes:

```
public class Coin
{
   public Coin(double aValue, String aName) { ... }
   public double getValue() { ... }
   ...
}

public class CashRegister
{
   public void enterPayment(int coinCount, Coin coinType)
      { ... }
   ...
}
```

Coupling

- A class *depends* on another if it uses objects of that class
- `CashRegister` depends on `Coin` to determine the value of the payment
- `Coin` does not depend on `CashRegister`
- High coupling = Many class dependencies
- Minimize coupling to minimize the impact of interface changes
- To visualize relationships draw class diagrams
- UML: Unified Modeling Language
  - *Notation for object-oriented analysis and design*

Dependency

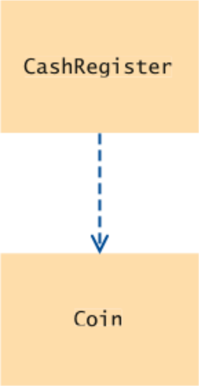

Figure 1
Dependency Relationship Between the CashRegister and Coin Classes

High and Low Coupling Between Classes

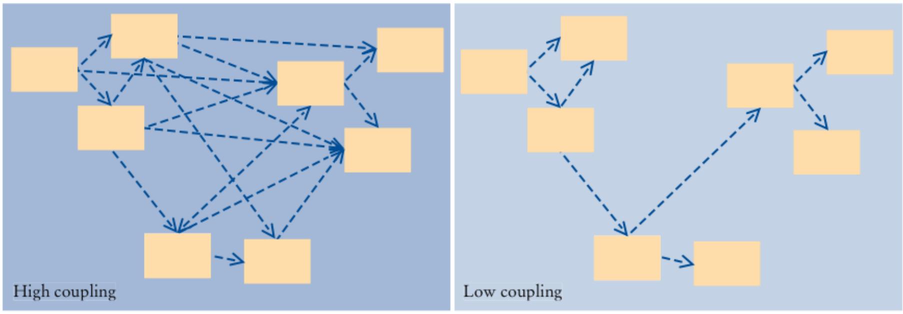

Figure 2 High and Low Coupling Between Classes

Self Check 8.3

Why is the `CashRegister` class from Chapter 4 not cohesive?

Answer: Some of its features deal with payments, others with coin values.

Self Check 8.4

Why does the `Coin` class not depend on the `CashRegister` class?

Answer: None of the `Coin` operations require the `CashRegister` class.

Immutable Classes

- **Accessor:** Does not change the state of the implicit parameter:

```
double balance = account.getBalance();
```

- **Mutator:** Modifies the object on which it is invoked:

```
account.deposit(1000);
```

- **Immutable class:** Has no mutator methods
- **Example**: `String`:

```
 String name = "John Q. Public";
 String uppercased = name.toUpperCase();
// name is not changed
```

Big Java by Cay Horstmann

Self Check 8.6

Is the `substring` method of the `String` class an accessor or a mutator?

> **Answer:** It is an accessor — calling `substring` doesn’t modify the string on which the method is invoked. In fact, all methods of the `String` class are accessors.

Big Java by Cay Horstmann

Side Effects

- This method has the expected side effect of modifying the implicit parameter and the explicit parameter `other`:

```
public void transfer(double amount, BankAccount other
{
   balance = balance - amount;
   other.balance = other.balance + amount;
}
```

Side Effects

- Another example of a side effect is output:

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $"
       + balance);
}
```

  Bad idea: Message is in English, and relies on `System.out`

- Decouple input/output from the actual work of your classes
- Minimize side effects that go beyond modification of the implicit parameter

Call by Value and Call by Reference

- **Call by value:** Method parameters are copied into the parameter variables when a method starts
- **Call by reference:** Methods can modify parameters
- Java has call by value
- A method can change state of object reference parameters, but cannot replace an object reference with another

Call by Value and Call by Reference

```
public class BankAccount
{
   public void transfer(double amount, BankAccount
      otherAccount)
   {
      balance = balance - amount;
      double newBalance = otherAccount.balance + amount;
      otherAccount = new BankAccount(newBalance);
      // Won't work
   }
}
```

Scope of Local Variables

- **Scope of variable:** Region of program in which the variable can be accessed
- Scope of a local variable extends from its declaration to end of the block that encloses it

Scope of Local Variables

- Sometimes the same variable name is used in two methods:

```
public class RectangleTester
{
   public static double area(Rectangle rect)
   {
      double r = rect.getWidth() * rect.getHeight();
      return r;
   }
   public static void main(String[] args)
   {
      Rectangle r = new Rectangle(5, 10, 20, 30);
      double a = area(r);
      System.out.println(r);
   }
}
```

- These variables are independent from each other; their scopes are disjoint

Scope of Local Variables

- Scope of a local variable cannot contain the definition of another variable with the same name:

```
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{
   double r = Math.sqrt(x);
   // Error - can't declare another variable
  // called r here
   ...
}
```

Big Java by Cay Horstmann

Scope of Local Variables

- However, can have local variables with identical names if scopes do not overlap:

```
if (x >= 0)
{
   double r = Math.sqrt(x);
   ...
   } // Scope of r ends here
else
{
   Rectangle r = new Rectangle(5, 10, 20, 30);
   // OK - it is legal to declare another r here
   ...
}
```

Big Java by Cay Horstmann

Overlapping Scope

- A local variable can *shadow* a variable with the same name
- Local scope wins over class scope:

```
public class Coin
{
   ...
   public double getExchangeValue(double exchangeRate)
   {
      double value; // Local variable
      ...
      return value;
   }
   private String name;
   private double value; // variable with the same name
}
```

Overlapping Scope

- Access shadowed variables by qualifying them with the `this` reference:

```
value = this.value * exchangeRate;
```

Overlapping Scope

- Generally, shadowing an instance variable is poor code — error-prone, hard to read
- Exception: when implementing constructors or setter methods, it can be awkward to come up with different names for instance variables and parameters
- OK:

```
public Coin(double value, String name)
{
   this.value = value;
   this.name = name;
}
```

Self Check 8.16

Consider the following program that uses two variables named `r`. Is this legal?

```
public class RectangleTester
{
   public static double area(Rectangle rect)
   {
      double r = rect.getWidth() * rect.getHeight();
      return r;
   }
   public static void main(String[] args)
   {
      Rectangle r = new Rectangle(5, 10, 20, 30);
      double a = area(r);
      System.out.println(r);
   }
}
```

Answer: Yes. The scopes are disjoint.

Self Check 8.17

What is the scope of the `balance` variable of the `BankAccount` class?

> **Answer:** It starts at the beginning of the class and ends at the end of the class.

Big Java by Cay Horstmann