Java Program Design - Anotações Ch. 4

Data: 06/04/2023
Tags: #SoftwareEngineering #java #POO
PDF: Java Program Design Principles, Polymorphism, and Patterns (Edward Sciore).pdf

---

Strategy heirarchies are a central component of several design techniques.

The Strategy Pattern

```
public abstract class IntProcessor {
        public void operateOn (int x) {´
                int y = f(x);
                System.out.println(x + " becomes " + y);
        }

        protected abstract int f (int x);
}

public class AddOne extends IntProcessor {
        protected int f (int x) {
                return x+1;
        }
}

public class AddTwo extends IntProcessor {
        protected int f (int x) {
                return x+2;
        }
}
```

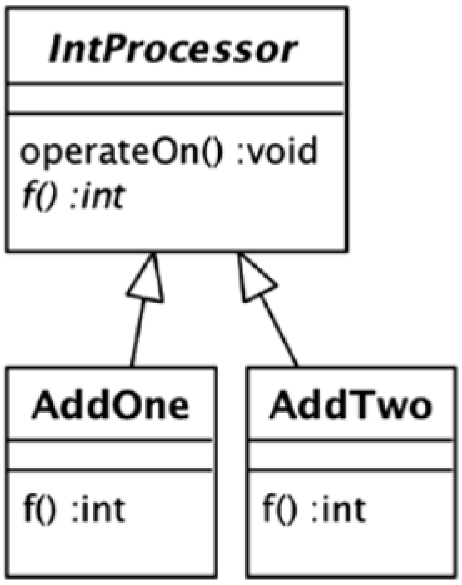
```
public class TestClient {
        public static void main (String[] args) {
                IntProcessor p1 = new AddOne();
                IntProcessor p2 = new AddTwo();
                p1.operateOn(6); // prints "6 becomes 7"
                p2.operateOn(6); // printf "6 becomes 8"
        }
}
```

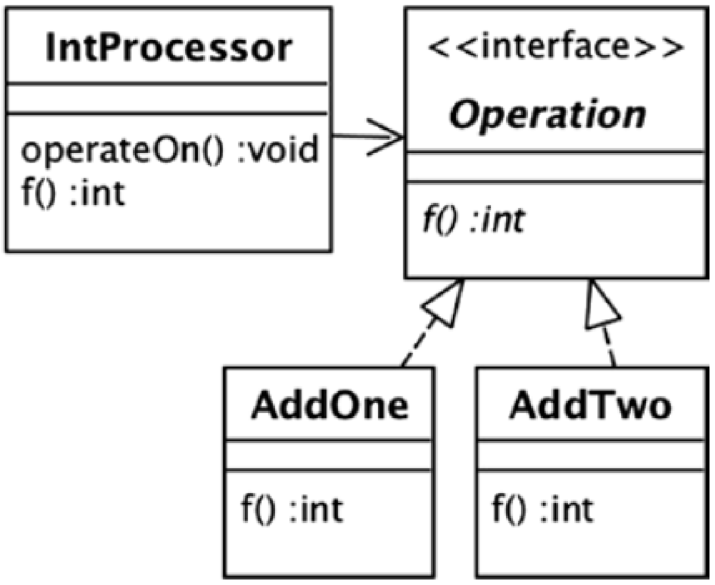
Another way to design this program is to not use subclassing. Instead of implementing the strategy classes as sublasses of IntProcessor, you can give them their own hierarchy, called a **strategy hierarchy.**

The hierarchy's interface is named Operation, and has the method f. The IntProcessor class, which no longer has any subclasses or abstract methods, holds a reference to an Operation object and uses that reference when it needs to call f. The TestClient class creates the desired Operation objects and passes each to IntProcessor via dependency injection.

```
public class IntProcessor {
        private Operation op;

        public IntProcessor (Operation op) {
                this.op = op;
        }

        public void operateOn (int x) {
                int y = f(x);
                System.out.println(x + " becomes " + y);
        }

        private int f (int x) {
                return op.f(x);
        }
}

interface Operation {
        public int f (int x);
}

class AddOne implements Operation {
        public int f (int x) {
                return x+1;
        }
}

class AddTwo implements Operation {
        public int f (int x) {
                return x+2;
        }
}
```

```
public class TestClient {
        public static void main (String[] args) {
                Operation op1 = new AddOne();
                Operation op2 = new AddTwo();
                IntProcessor p1 = new IntProcessor(op1);
                IntProcessor p2 = new IntProcessor(op2);
                p1.operateOn(6);
                p2.operateOn(6);
        }
}
```





The technique of organizing strategy classes into a hierarchy is called the *strategy pattern*. The strategy pattern is depicted by the class diagram of Figure 4-2. The strategy interface defines a set of methods. Each class that implements the interface provides a different strategy for performing those methods. The client has a variable that holds an object from one of the strategy classes. Because the variable is of type `StrategyInterface`, the client has no idea which class the object belongs to and consequently does not know which strategy is being used.

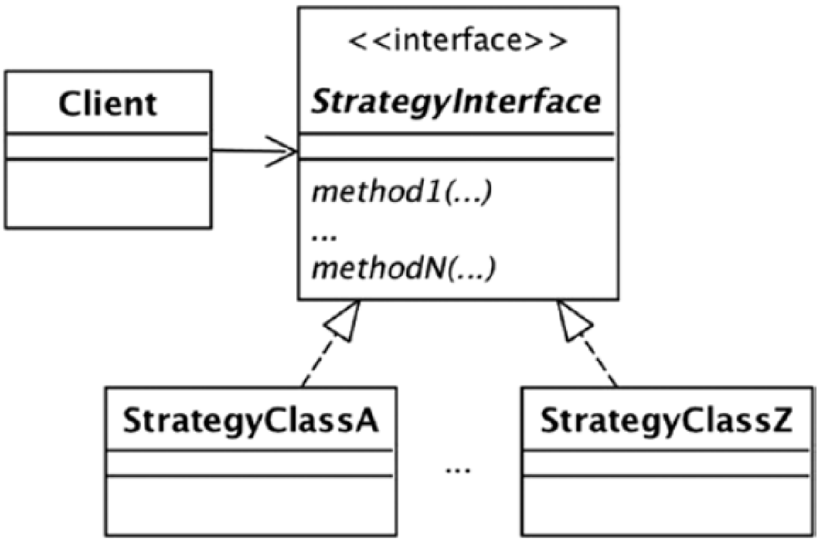


Figure 4-2. *The strategy pattern*

Comparators

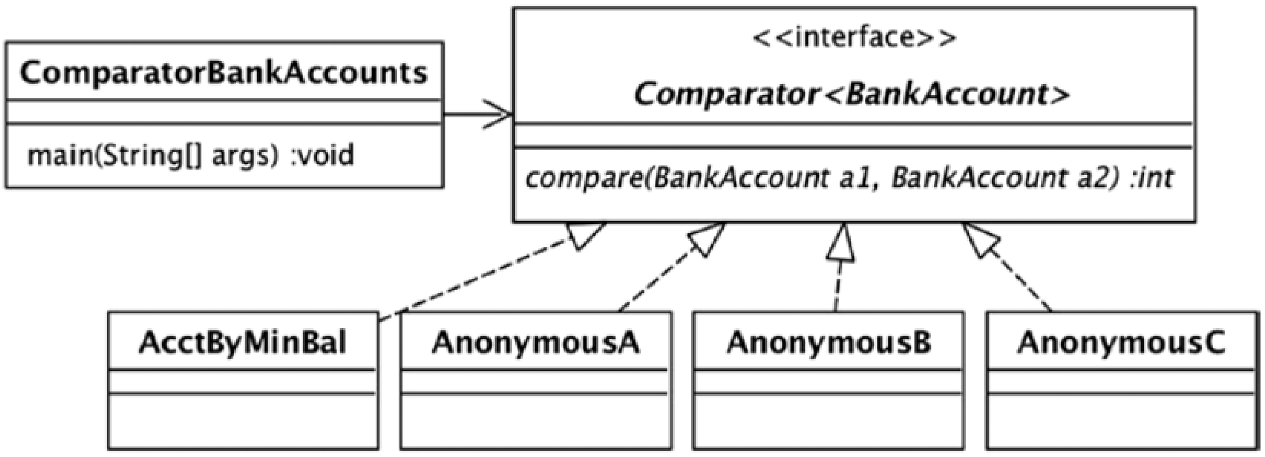
The problem is that Comparable hardcodes a specific ordering, which makes it essentially impossible to compare objects in any other way

How can you specify **different comparison orders**? Use the **strategy pattern**! The strategy interface declares the comparison method, and the strategy classes provide specific implementations of that method.

Because object comparison is so common, the Java library provides this strategy interface for you. The interface is called Comparator and the method it declares is called compare. The compare method is similar to compareTo except that it takes two objects as parameters. The call compare(x,y) returns a value greater than 0 if x>y, a value less than 0 if x

```
class AcctByMinBal implements Comparator<BankAccount> {
        public int compare (BankAccount ba1, BankAccount ba2) {
                int bal1 = ba1.getBalance();
                int bal2 = ba2.getBalance();
                if (bal1 == bal2)
                        return ba1.getAcctNum() - ba2.getAcctNum();
                else
                        return bal2 - bal1;
        }
}
```

```
public class ComparatorBankAccounts {
        public static void main (String[] args) {
                List accts = initAccts();
                Comparator minbal = new AcctByMinBal();
                Comparator maxbal = innerClassComp();
                Comparator minnum = lambdaExpComp1();
                Comparator maxnum = lambdaExpComp2();

                BankAccount a1 = findMax(accts, minbal);
                BankAccount a2 = findMax(accts, maxbal);
                BankAccount a3 = Collections.max(accts, minnum);
                BankAccount a4 = Collections.max(accts, maxnum);

                System.out.println("Acct with smallest bal is " + a1);
                System.out.println("Acct with largest bal is " + a2);
                System.out.println("Acct with smallest num is " + a3);
                System.out.println("Acct with largest num is " + a4);
        }
}
```

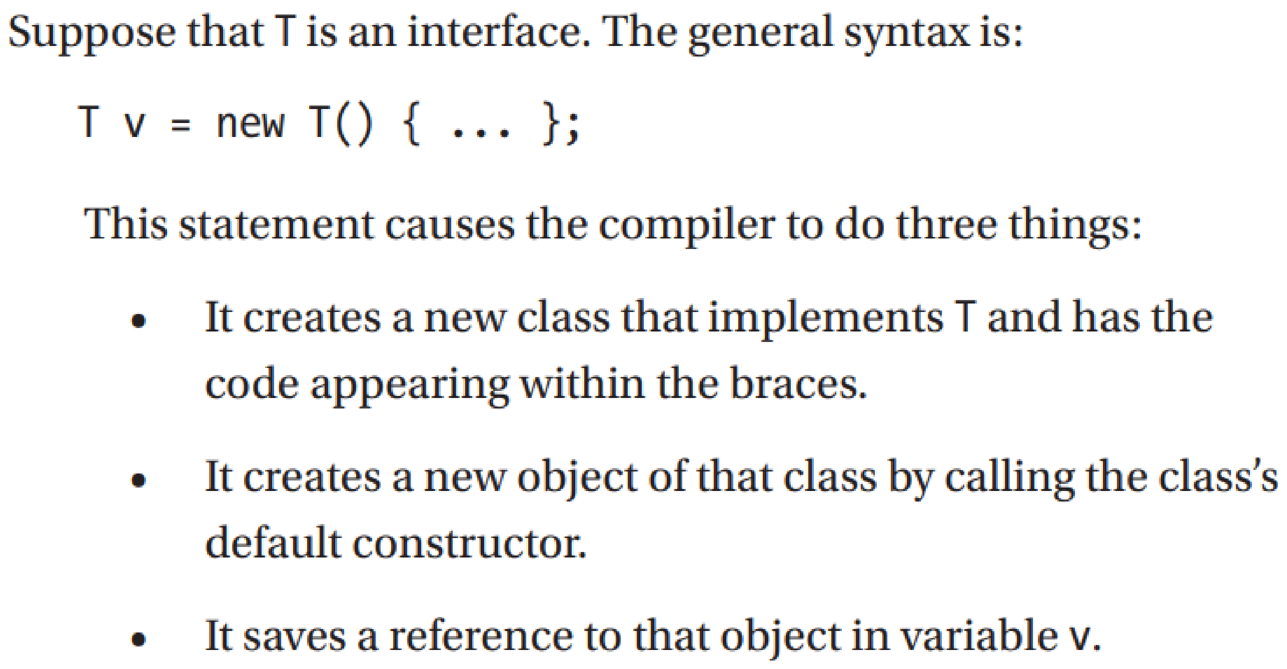


Anonymous Inner Classes

The rule of Abstraction asserts that the type of a variable should be an interface when possible. In such case them name of the class that implements the interface is relatively unimportant, as it will only be used when the class constructor is called.

Explicit Anonymous Classes

An anonymous inner class defines a class without giving it a name.


Suppose that T is an interface. The general syntax is:

```
T v = new T() { ... };
```

This statement causes the compiler to do three things:

- It creates a new class that implements T and has the code appearing within the braces.
- It creates a new object of that class by calling the class's default constructor.
- It saves a reference to that object in variable v.

```
private static Comparator innerClassComp() {
        Comparator result = new Comparator() {
                public int compare(BankAccount ba1, BankAccount ba2) {
                        int bal1 = ba1.getBalance();
                        int bal2 = ba2.getBalance();
                        if (bal1 == bal2) return ba1.getAcctNum() - ba2.getAcctNum();
                        else return bal1 - bal2;
                }
        };
        return result;
}
```

Lambda Expressions

```
private static Comparator innerClassComp() {
        Comparator result =
                (BankAccount ba1, BankAccount ba2) -> {
                        return ba2.getAcctNum() - ba1.getAcctNum();
                };
        return result;
}
```

Or even just:

```
private static Comparator innerClassComp() {
        Comparator result =
                (ba1, ba2) -> ba2.getAcctNum() - ba1.getAcctNum();
        return result;
}
```

Although lambda expressions can be written reasonably compactly, Java lets you abbreviate them even further.

- You don't have to specify the types of the parameters.
- If there is only one parameter then you can omit the parentheses around it.
- If the body of the method consists of a single statement then you can omit the braces; if a single-statement method also returns something then you also omit the "return" keyword.

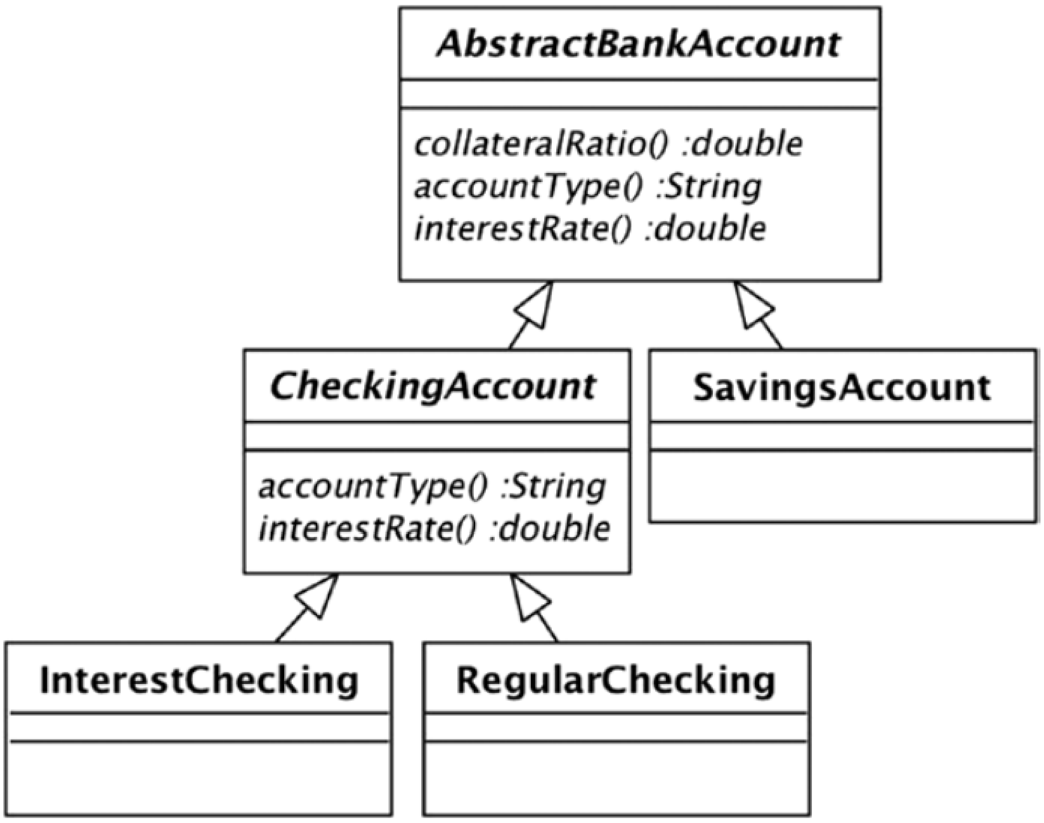
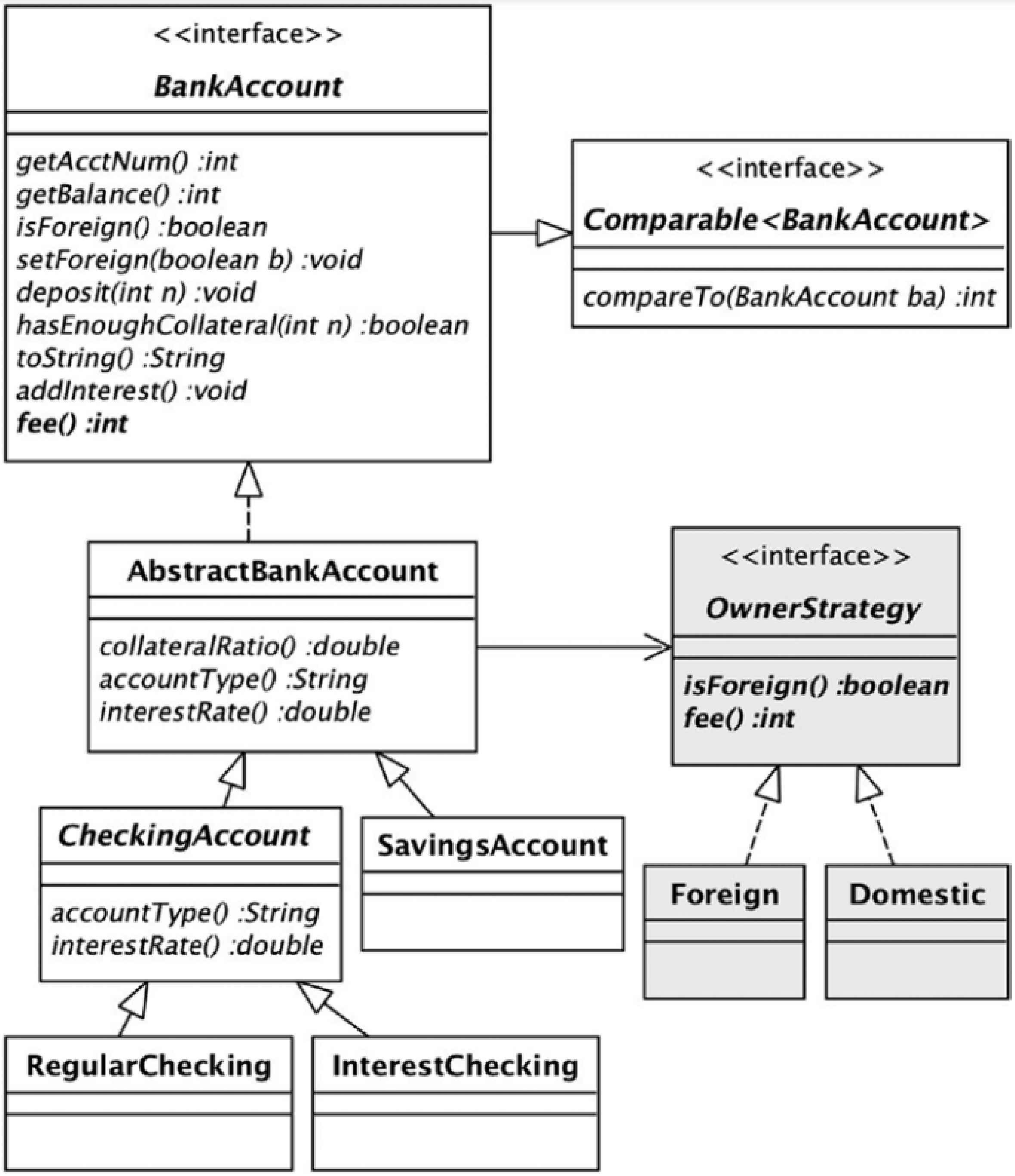


Figure 4-6. *The version 9 BankAccount hierarchy*



The command pattern

```
private void processCommand(int cnum) {
   InputCommand cmd = commands[cnum];
   current = cmd.execute(scanner, bank, current);
   if (current < 0)
      done = true;
}
```

The strategy interface `InputCommand` has eight implementing classes—one class for each type of command. These classes are named `QuitCmd`, `NewCmd`, `DepositCmd`, and so on. Figure 4-8 shows their class diagram.

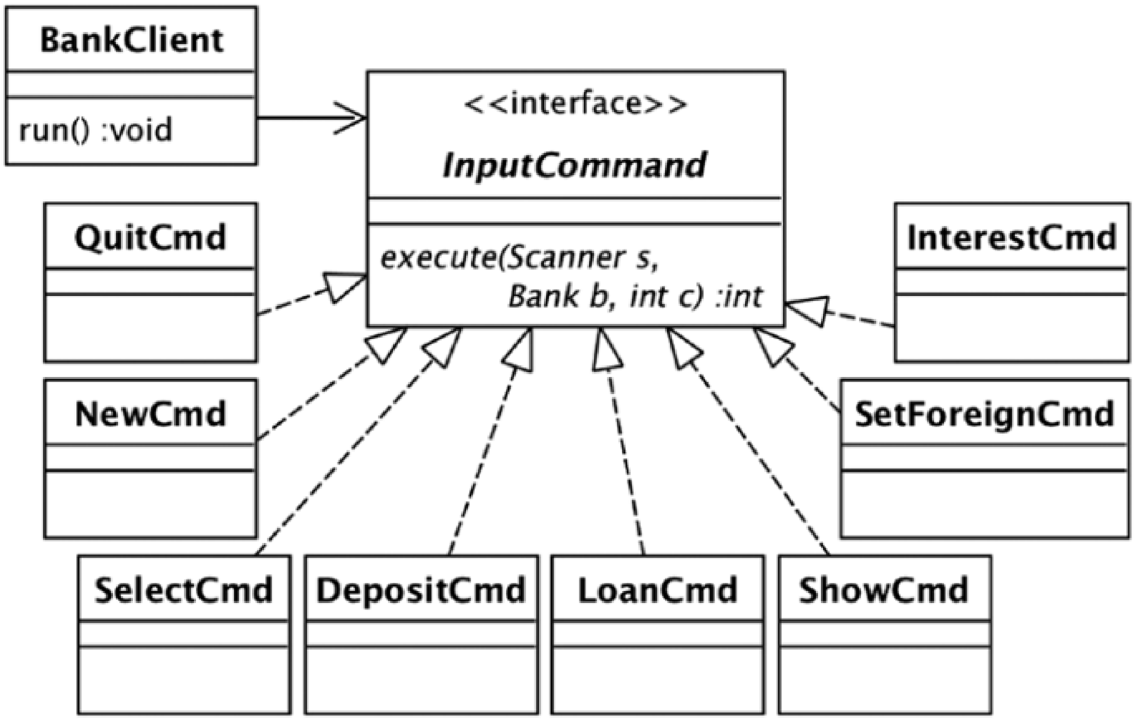


Figure 4-8. *The InputCommand Strategy Hierarchy*

Listing 4-18. The Version 10 InputCommand Interface

```
public interface InputCommand {
   int execute(Scanner sc, Bank bank, int current);
}
```

Listing 4-19. The Version 10 DepositCmd Class

```
public class DepositCmd implements InputCommand {
   public int execute(Scanner sc, Bank bank, int current) {
      System.out.print("Enter deposit amt: ");
      int amt = sc.nextInt();
      bank.deposit(current, amt);
```

```
public class BankClient {
   private Scanner scanner;
   private boolean done = false;
   private Bank bank;
   private int current = 0;
   private InputCommand[] commands = {
         new QuitCmd(),
         new NewCmd(),
         new SelectCmd(),
         new DepositCmd(),
         new LoanCmd(),
         new ShowCmd(),
         new InterestCmd(),
         new SetForeignCmd() };

   public BankClient(Scanner scanner, Bank bank) {
      this.scanner = scanner;
      this.bank = bank;
   }

   public void run() {
      String usermessage = constructMessage();
      while (!done) {
         System.out.print(usermessage);
         int cnum = scanner.nextInt();
         processCommand(cnum);
      }
```

```
    private String constructMessage() {
       int last = commands.length-1;
       String result = "Enter Account Type (";
       for (int i=0; i<last; i++)
          result += i + "=" + commands[i] + ", ";
       result += last + "=" + commands[last] + "): ";
       return result;
    }

    private void processCommand(int cnum) {
       InputCommand cmd = commands[cnum];
       current = cmd.execute(scanner, bank, current);
       if (current < 0)
          done = true;
    }
 }
```

Eliminating the Class Hierarchy

The duality between the template pattern and the strategy pattern implies that any design using the template pattern can be redesigned to use the strategy pattern. This section shows how to redesign the banking demo so that its `BankAccount` class hierarchy is replaced by a strategy hierarchy. This redesign is version 11 of the banking demo.

The idea of the redesign is to implement `SavingsAccount`, `RegularChecking`, and `InterestChecking` as strategy classes, headed by a strategy interface named `TypeStrategy`. The interface declares the three methods `collateralRatio`, `accountType`, and `interestRate`. Consequently, `AbstractBankAccount` will no longer need subclasses. Instead, it will implement these three methods via its reference to a `TypeStrategy` object.

Figure 4-9 shows the version 11 class diagram. In this design, `AbstractBankAccount` has two strategy hierarchies. The `OwnerStrategy` hierarchy is the same as in version 10. The `TypeStrategy` hierarchy contains the code for the methods of `AbstractBankAccount` that were previously abstract.

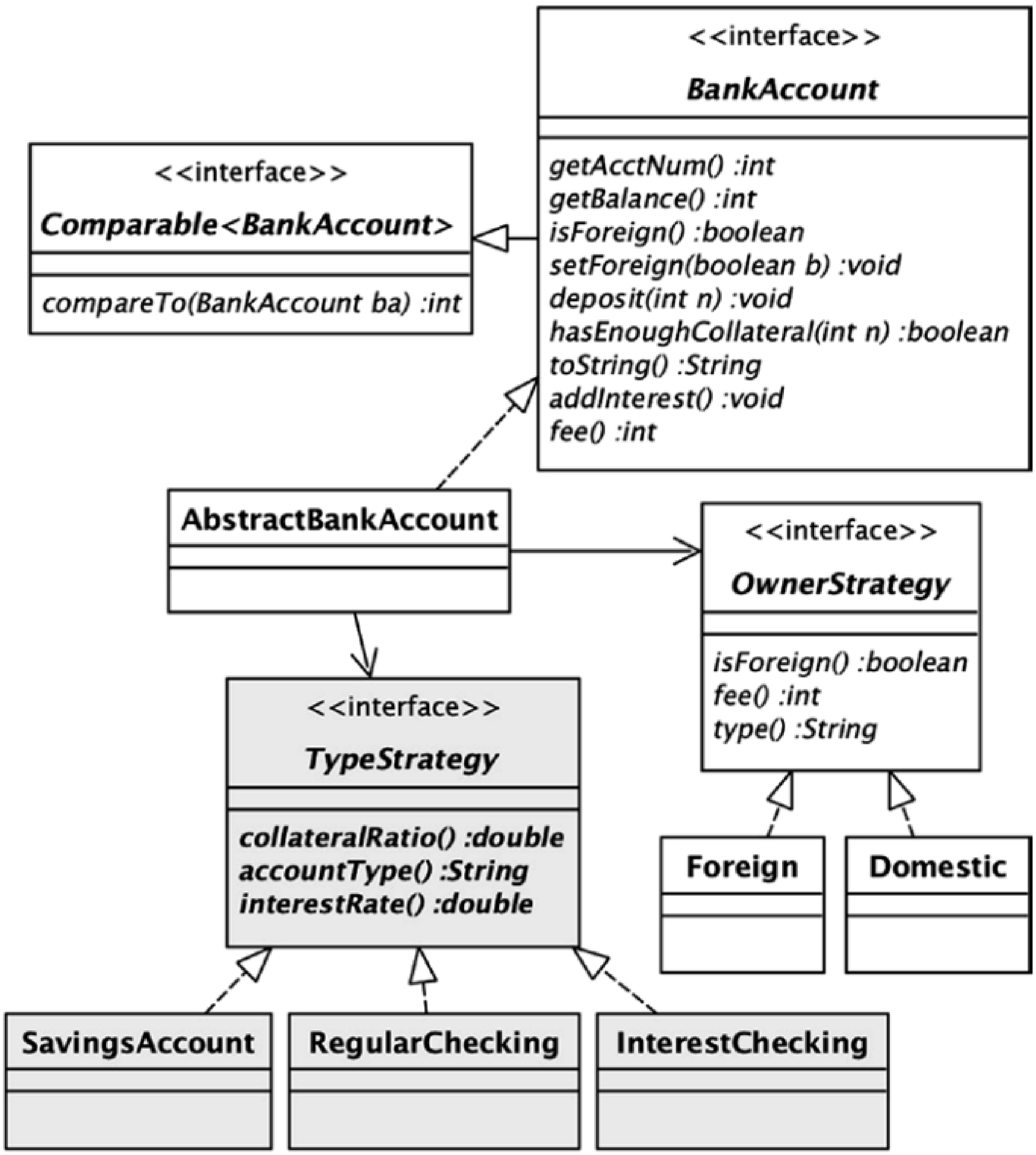


Figure 4-9. *Version 11 of the Banking Demo*

```
public interface TypeStrategy {
   double collateralRatio();
   String accountType();
   double interestRate();
}
```

The classes SavingsAccount, RegularChecking, and InterestChecking implement TypeStrategy. These classes are essentially unchanged from version 10; the primary difference is that they now implement TypeStrategy instead of extending AbstractBankAccount. Listing 4-23 gives the code for SavingsAccount; the code for the other two classes is similar.

Listing 4-23. The Version 11 SavingsAccount Class

```
public class SavingsAccount implements TypeStrategy {
   public double collateralRatio() {
      return 1.0 / 2.0;
   }

   public String accountType() {
      return "Savings";
   }

   public double interestRate() {
      return 0.01;
   }
}
```

```
public int newAccount(int type, boolean isforeign) {
   int acctnum = nextacct++;
   TypeStrategy ts;
   if (type==1)
      ts = new SavingsAccount();
   else if (type==2)
      ts = new RegularChecking();
   else
      ts = new InterestChecking();
   BankAccount ba = new AbstractBankAccount(acctnum, ts);
   ba.setForeign(isforeign);
   accounts.put(acctnum, ba);
   return acctnum;
}
```

```
public class AbstractBankAccount implements BankAccount {
   private int acctnum;
   private int balance = 0;
   private OwnerStrategy owner = new Domestic();
   private TypeStrategy ts;

   public AbstractBankAccount(int acctnum, TypeStrategy ts) {
      this.acctnum = acctnum;
      this.ts = ts;

   }
   ...
   private double collateralRatio() {
      return ts.collateralRatio();
   }

   private String accountType() {
      return ts.accountType();
   }

   private double interestRate() {
      return ts.interestRate();
   }
}
```