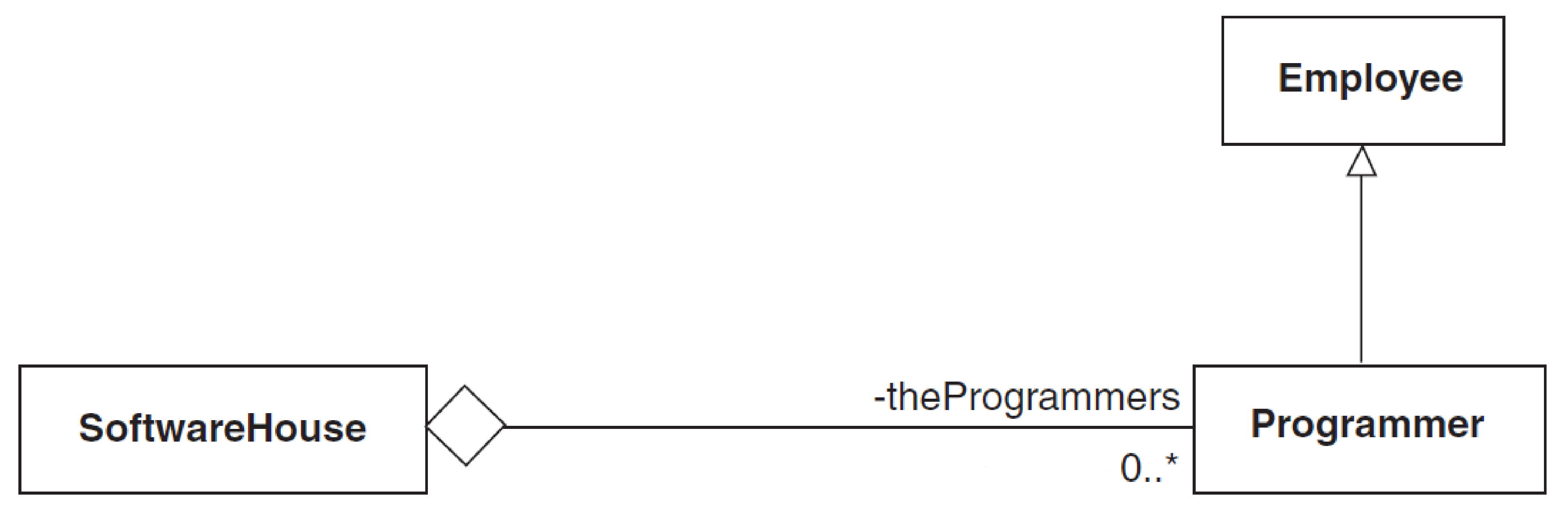
05 – Inheritance

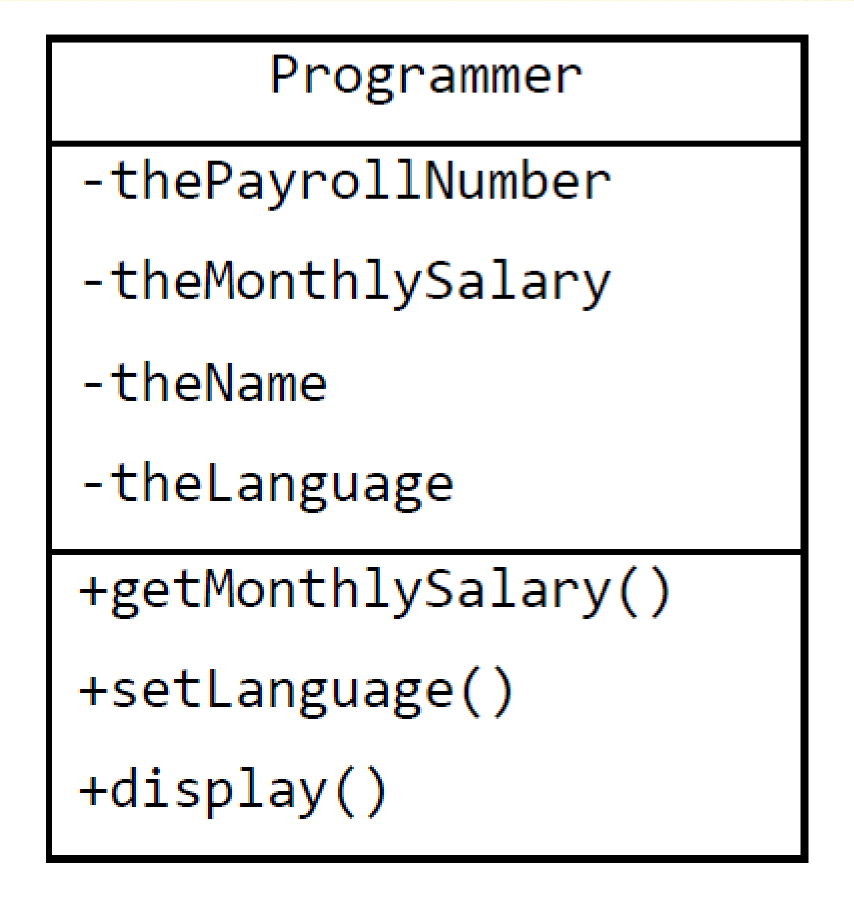
OBJECT-ORIENTED PROGRAMMING

Specialization

- relationship that may also exist between classes
  - association, aggregation, and composition
  - specialization



The initial Programmer Class



Programmer

-thePayrollNumber

-theMonthlySalary

-theName

-theLanguage

+getMonthlySalary()

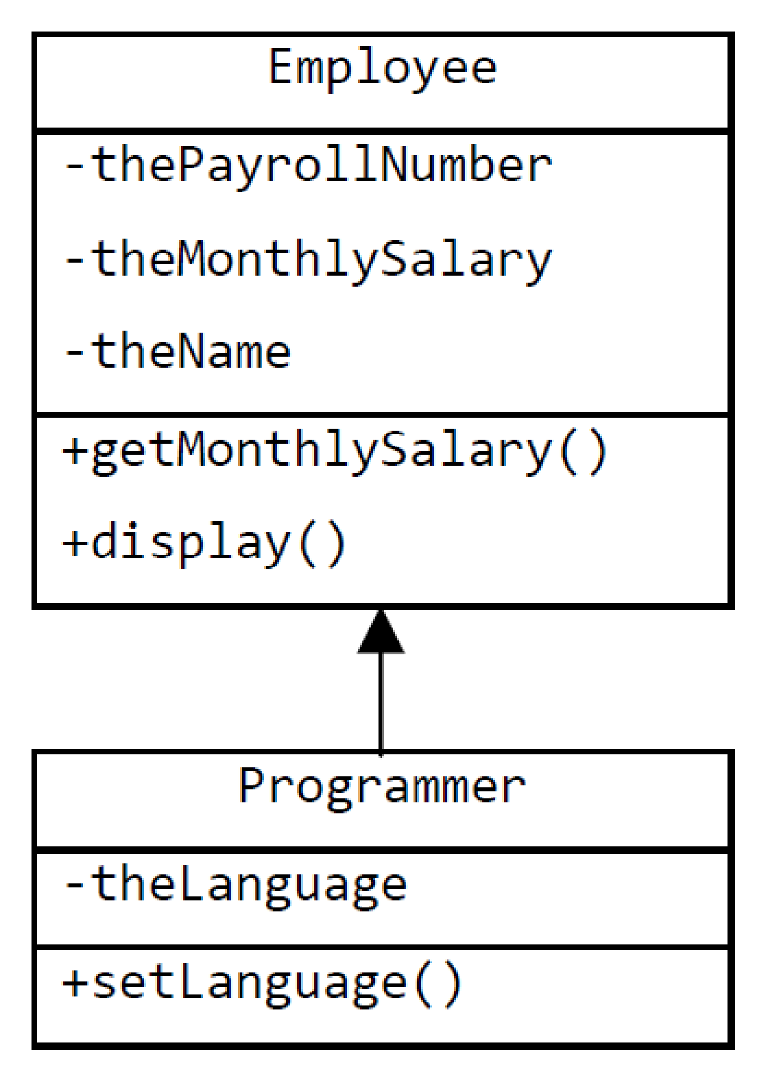
+setLanguage()

+display()

Employee Class

- All of the software house employees, not just programmers, are given a payroll number, a name and a salary
- The only attribute special to a Programmer is theLanguage
- A Programmer as a special kind of Employee
- Employee:
  - Parent, super, base class
- Programmer:
  - Child, sub, derived class

The Employee, Programmer Class



Defining A Subclass In C++

```
class Employee {
   int thePayrollNumber;
   int theMonthlySalary;
   string theName;
public:
   Employee(int aPayrollNumber, int aMonthlySalary, string aName) {
   //... }
   int getMonthlySalary() { //... }
   void display() { //... }
}

class Programmer: public Employee{
   string theLanguage;
public:
   Programmer(int aPayrollNumber, int aMonthlySalary, string aName,
   string aLanguage) : Employee(aPayrollNumber, aMonthlySalary,
   aName) { //... }
   int setLanguage(string aLanguage) { //... }
}
```

Defining a Subclass In Java

```
class Employee {
   private int thePayrollNumber;
   private int theMonthlySalary;
   private String theName;
   public Employee(int aPayrollNumber, int aMonthlySalary, String
   aName) { //... }
   public int getMonthlySalary() { //... }
   public void display() { //... }
}

class Programmer extends Employee{
   private String theLanguage;
   public Programmer(int aPayrollNumber, int aMonthlySalary, String
   aName, String aLanguage) {
        super(aPayrollNumber, aMonthlySalary, aName);
        //... }
   public int setLanguage(String aLanguage) { //... }
}
```

Access Control For Class Members

- To prevent direct alteration of a data member or direct invocation of a member function
- Every member of a class has associated with it an access control property
- In C++, a member can be **`private`**, **`protected`**, or **`public`**
- In addition, Java also allows for the access control property of a member to be package

Access Control In C++

- All members of a C++ class are **`private`** unless declared explicitly to be otherwise
- The **`public`** members of a class:
  - accessible to all other classes and functions
  - inherited by its subclasses
- The **`private`** members of a class:
  - accessible only to the definitions that are meant specifically for that class – **`friend`** functions
  - cannot be accessed within the subclasses
- The **`protected`** members of a class:
  - accessible to only the subclasses

Access Control In Java

- The modifiers **`public`** and **`private`** carry the same meaning as in C++.
- The modifier **`protected`** also carries the same meaning, except that such members are like public members in the same package
- When no access modifier is specified in Java, that means the member has access control of type **package**

Inherited Methods

- Methods and attributes declared in a superclass are inherited by a subclass
- Need only declare those methods and attributes special to the subclass
- **Programmer**: the attribute **theLanguage** and the method **setLanguage()**
- Example:

```
programmer.getMonthlySalary();
programmer.setLanguage(“Java”);
employee.setLanguage(“Java”);    // illegal
```

Redefined Methods

- Any **`private`** attributes in a superclass cannot be accessed in a subclass method
  - the method body for display in the **`Employee`** class cannot be modified by the **`Programmer`** class
- A method inherited by a subclass can be redefined to have a different behavior
  - make use of the **`display`** method in the **`Employee`** class

Redefined Methods in C++

```
// class Employee
void display() {
   cout << "Payroll Number:\t" << thePayrollNumber << "\n";
   cout << "Monthly Salary:\t" << theMonthlySalary << "\n";
   cout << "First Name:\t" << theName << "\n";
}

// class Programmer
void display() {
   Employee::display();
   cout << "The Language:\t" << theLanguage << "\n";
}
```

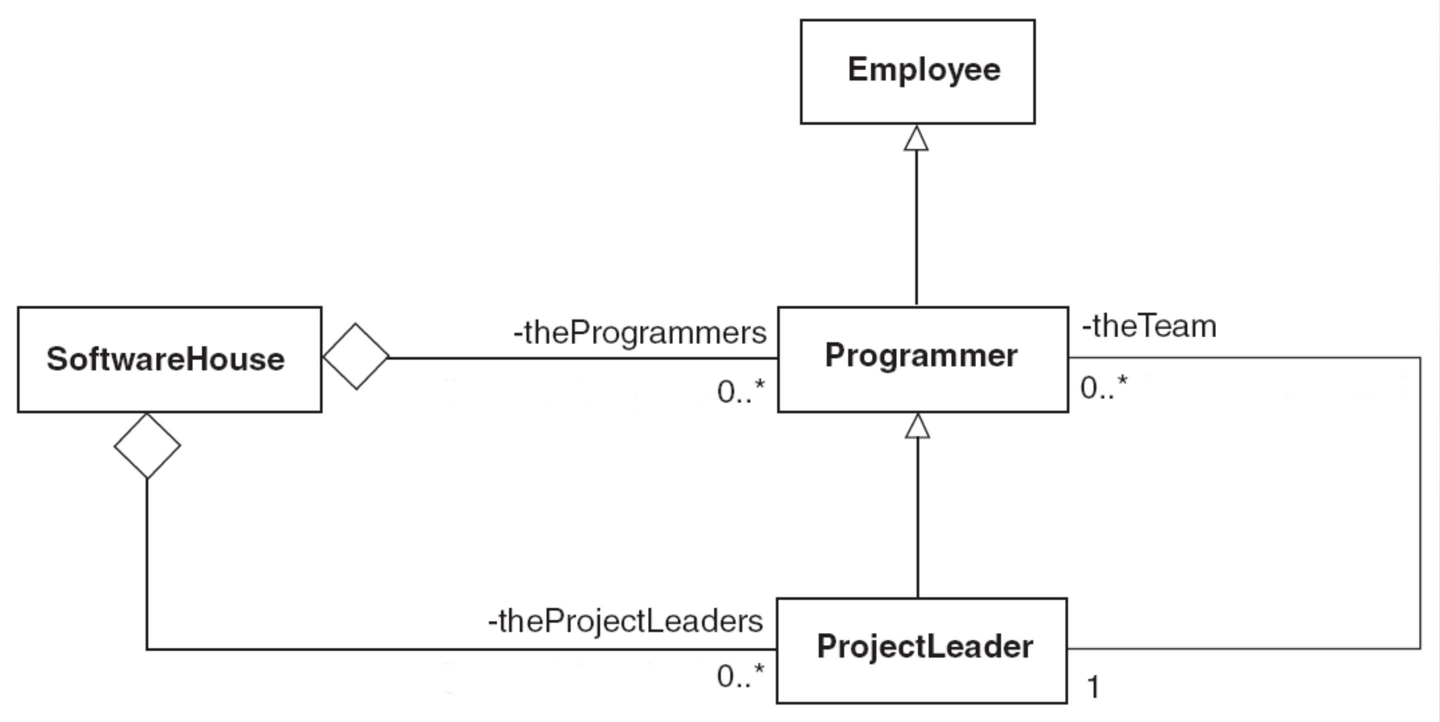
Redefined Methods in Java

```
// class Employee
void display() {
   System.out.println("Payroll Number:\t" +thePayrollNumber);
   System.out.println("Monthly Salary:\t" +theMonthlySalary);
   System.out.println("First Name:\t" + theName);
}

// class Programmer
void display() {
   super.display();
   System.out.println("The Language:\t" + theLanguage);
}
```

Polymorphism

- A message sent to an object of some class is received as normal.
- However the same message may also be received by an object of a descendant class.



The displayStaff Method

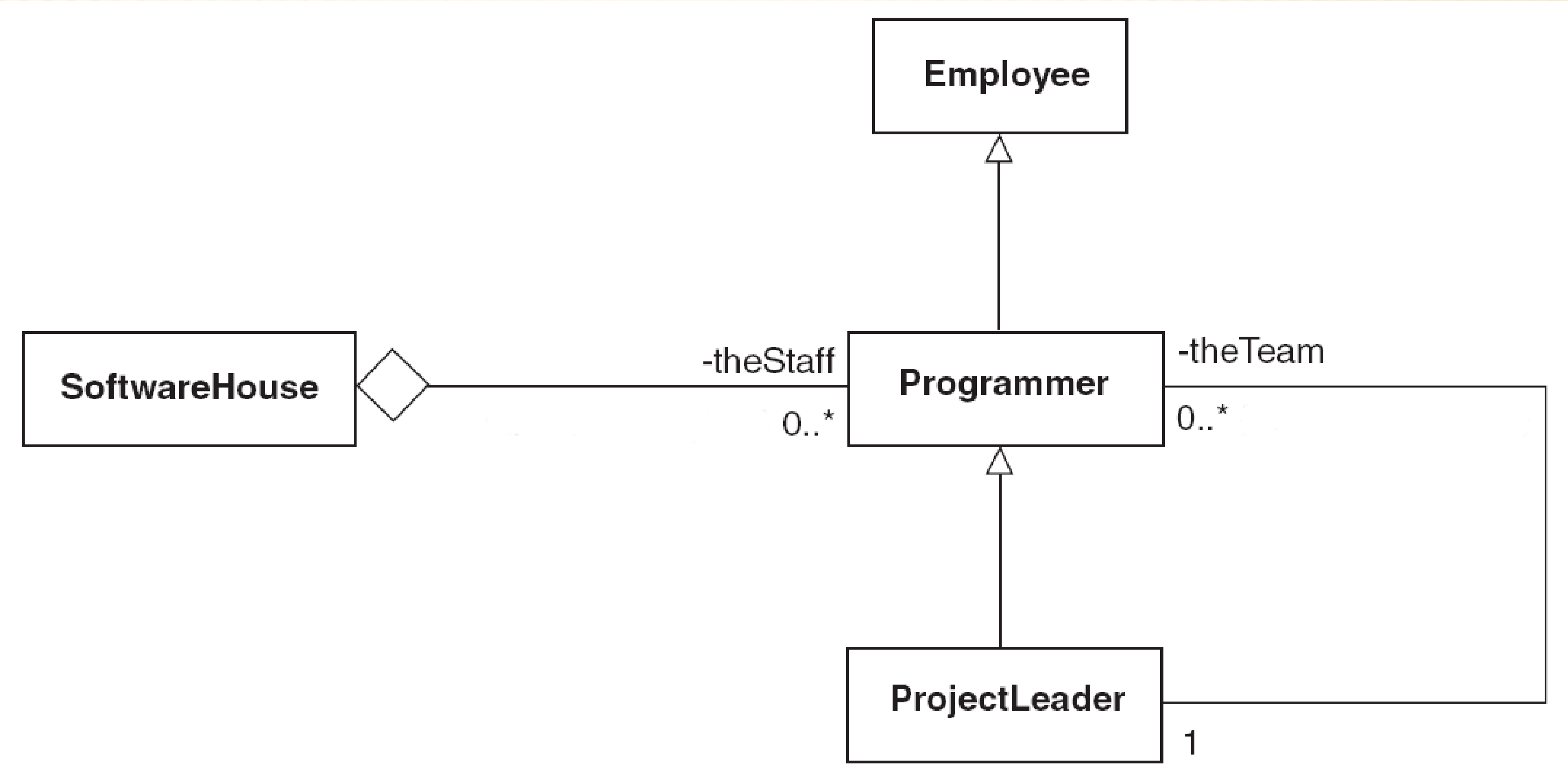
```
// pseudo code of class SoftwareHouse
void displayStaff() {
  for each programmer in theProgrammers
  {
     programmer.display();
  }
  for each projectLeader in theProjectLeaders
  {
     projectLeader.display();
  }
}
```

The Polymorphic Effect

- Code duplication is almost certainly unnecessary
- A child class must have at least the same set of methods as its parent
- An object of a child class can be used in place of an object of a parent class
  - a `ProjectLeader` object can substitute for a `Programmer` object
- **Late bingding**: the binding of a message to a corresponding method when the software is actually running

An Improved Class Diagram Using Polymorphism

- A message sent from a **SoftwareHouse** object through a reference/pointer to a **Programmer** object, can now be received by a **Programmer** object or a **ProjectLeader** object



The Simplified **displayStaff** Method

- There is no longer any need to have an explicit relationship between the **SoftwareHouse** and its **ProjectLeaders**
- The message **display** sent through a **Programmer** reference/pointer may be received by a **Programmer** or a **ProjectLeader** object

```
// pseudo code of class SoftwareHouse
void displayStaff() {
   for each programmer in theStaff
   {
      programmer.display();
   }
}
```

Example – 1 (pseudo code)

```
// main() method
   Create a new organisation.
   SoftwareHouse sh(“Objects-R-Us”);

   // Create some new programmers.
   Programmer p1(123, 2000, “John”, “Ada”) ;
   Programmer p2(234, 2500, “Ken”, “C++”);
   Programmer p3(456, 3000, “Peter”, “Java”) ;

   // Create some new project leaders.
   ProjectLeader pl1(567, 4000, “Jon”, “C”) ;
   ProjectLeader pl2(789, 4000, “Jessie”, “Java”);
```

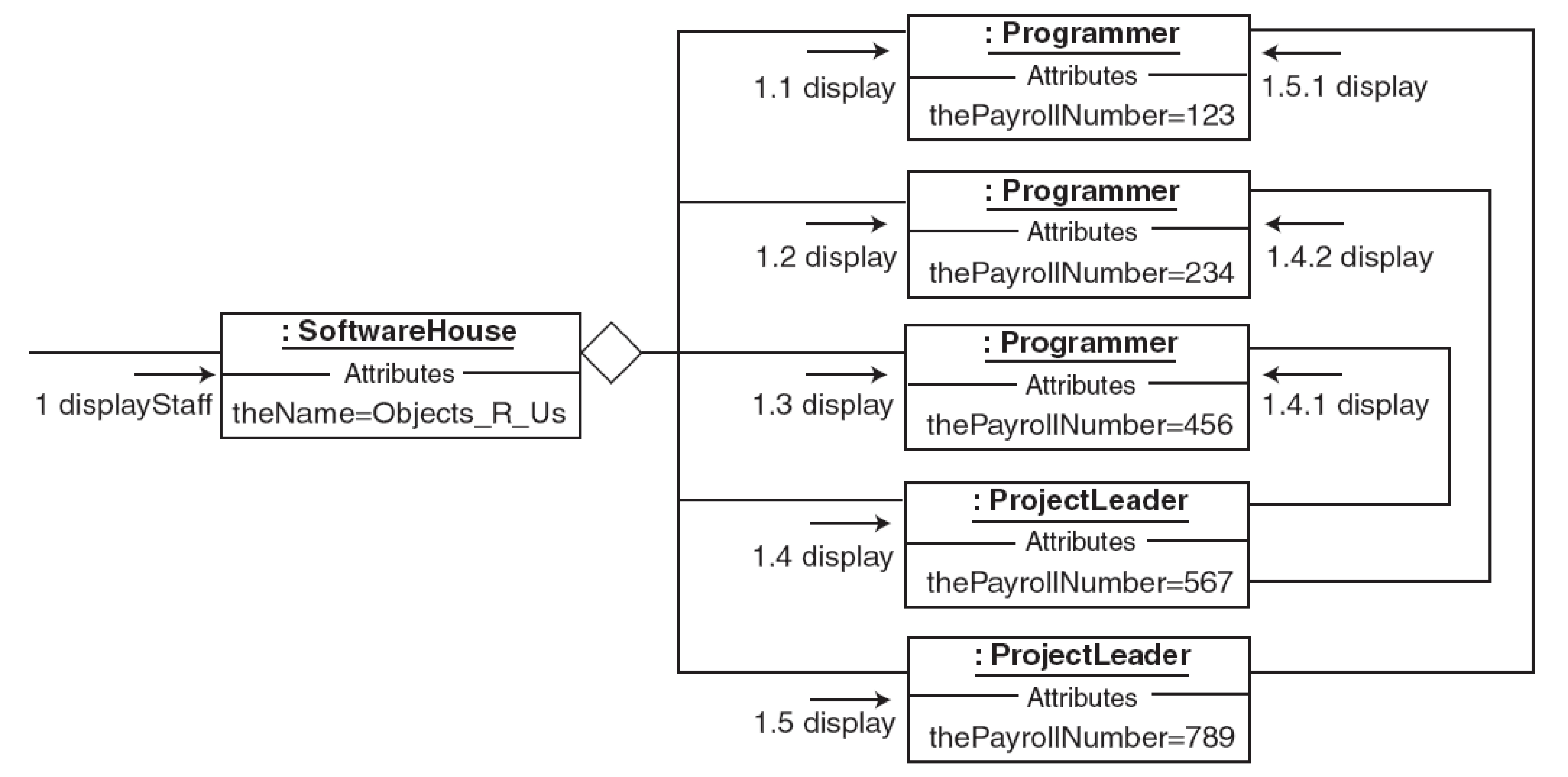
Example – 2

```
// main() method (cont.)
   // Assign each programmer to a project leader
   pl1.addProgrammer(p3);
   pl1.addProgrammer(p2);
   pl2.addProgrammer(p1);

   // Hire each programmer and project leader
   sh.addProgrammer(p1);
   sh.addProgrammer(p2);
   sh.addProgrammer(p3);
   sh.addProgrammer(pl1);
   sh.addProgrammer(pl2);

   // Display some details of the staff.
   sh.displayStaff();
```

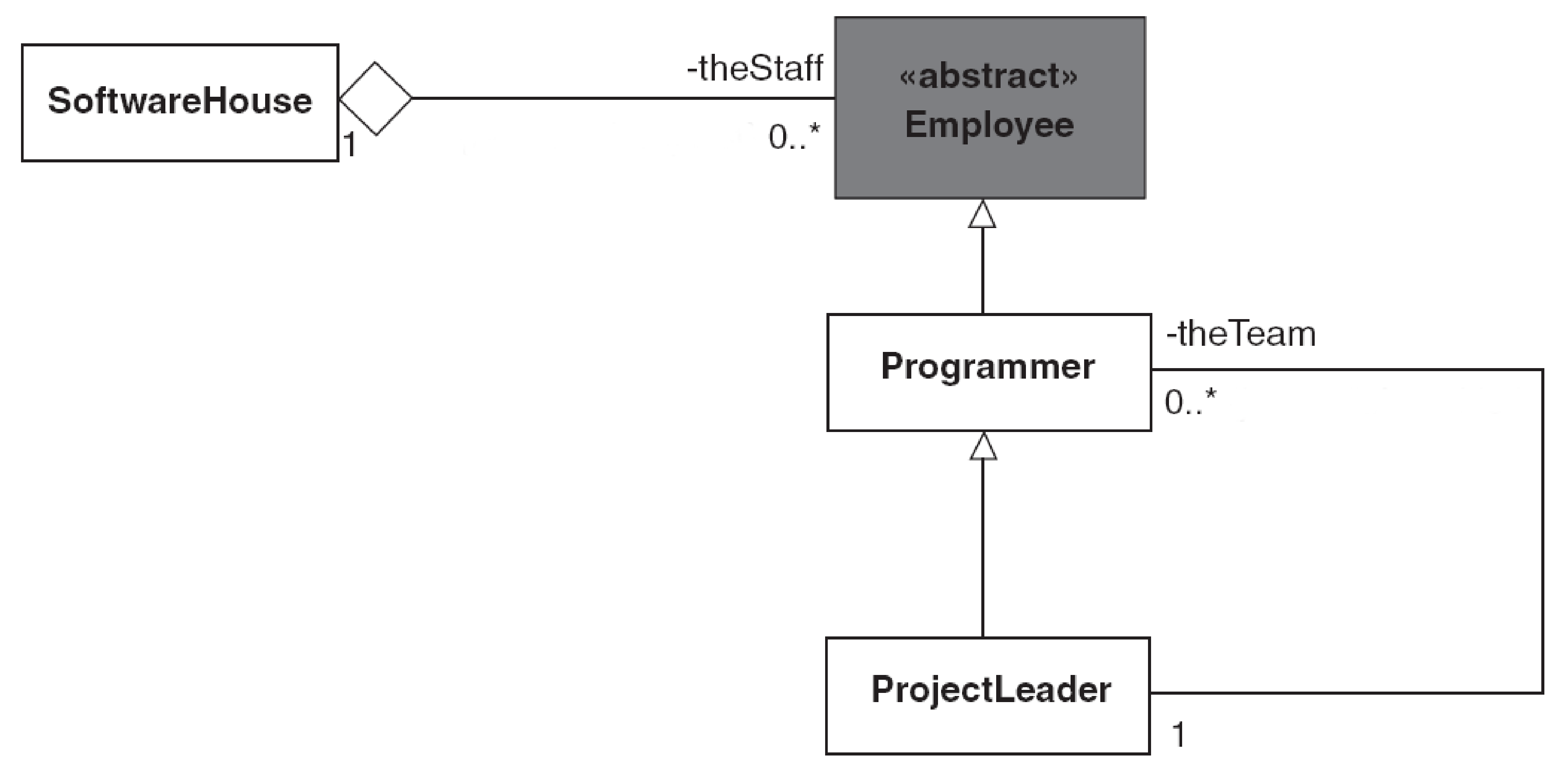
The Collaboration Diagram For The Operation Displaystaff



The **Abstract** Class

- A class that acts as a basis for establishing others
- No intention to make an instance of it
- All descendants share a common set of operations on their **public** interface
- Example:
    - there will never be an instance of an **Employee** as we have **Programmers** or **ProjectLeaders**
    - Abstract class **Employee**: share common operations such as **getPayrollNumber**, **getMonthlySalary**, and **display**

A Modified Class Diagram Using An Abstract Class



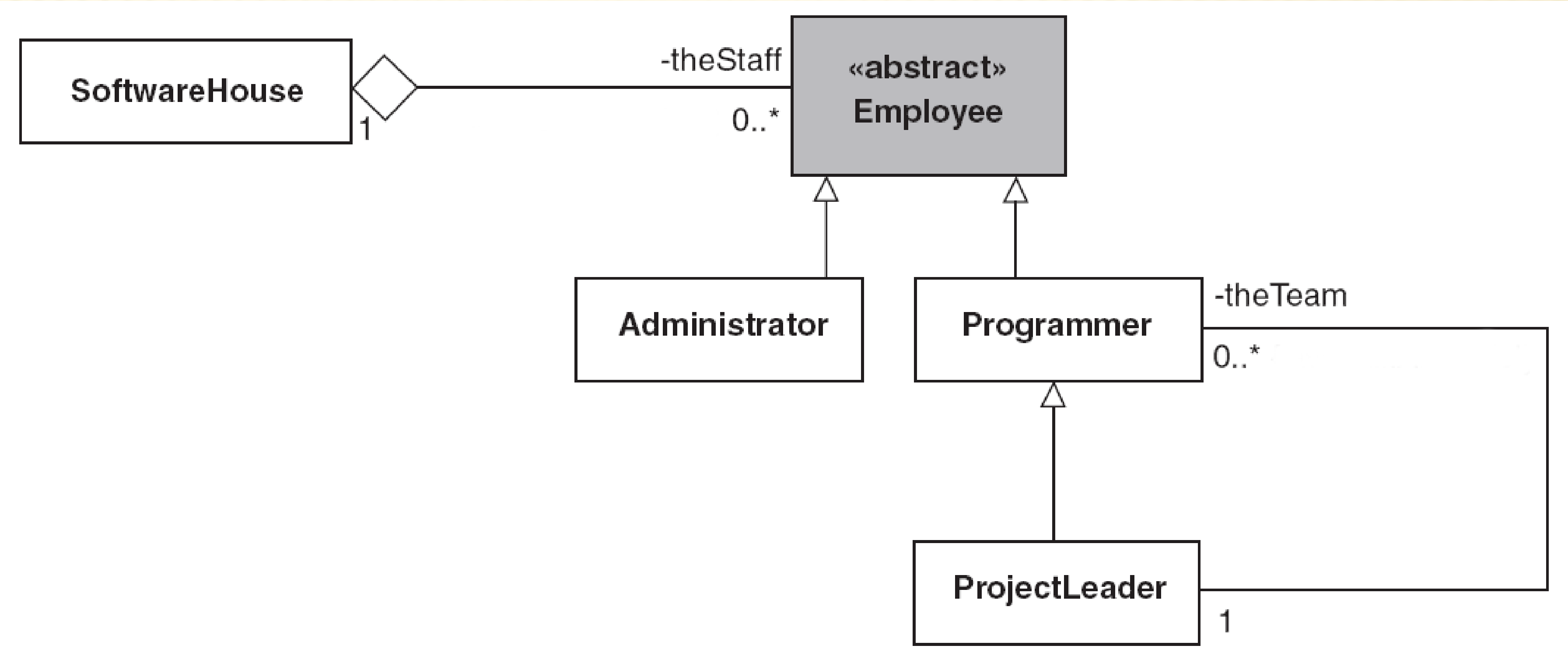
Changes In Code

```
// class SoftwareHouse
void addEmployee(Employee anEmployee)  {
  add anEmployee to theStaff
}

void displayStaff() {
  for each employee in theStaff
  {
      employee.display();
  }
}
```

A Modified Class Diagram

- introduce other kinds of Employees to the software house


SoftwareHouse
1
-theStaff
0..*
«abstract»
Employee
Administrator
Programmer
-theTeam
0..*
ProjectLeader
1