211lab09

Lab 09 – List Visitors

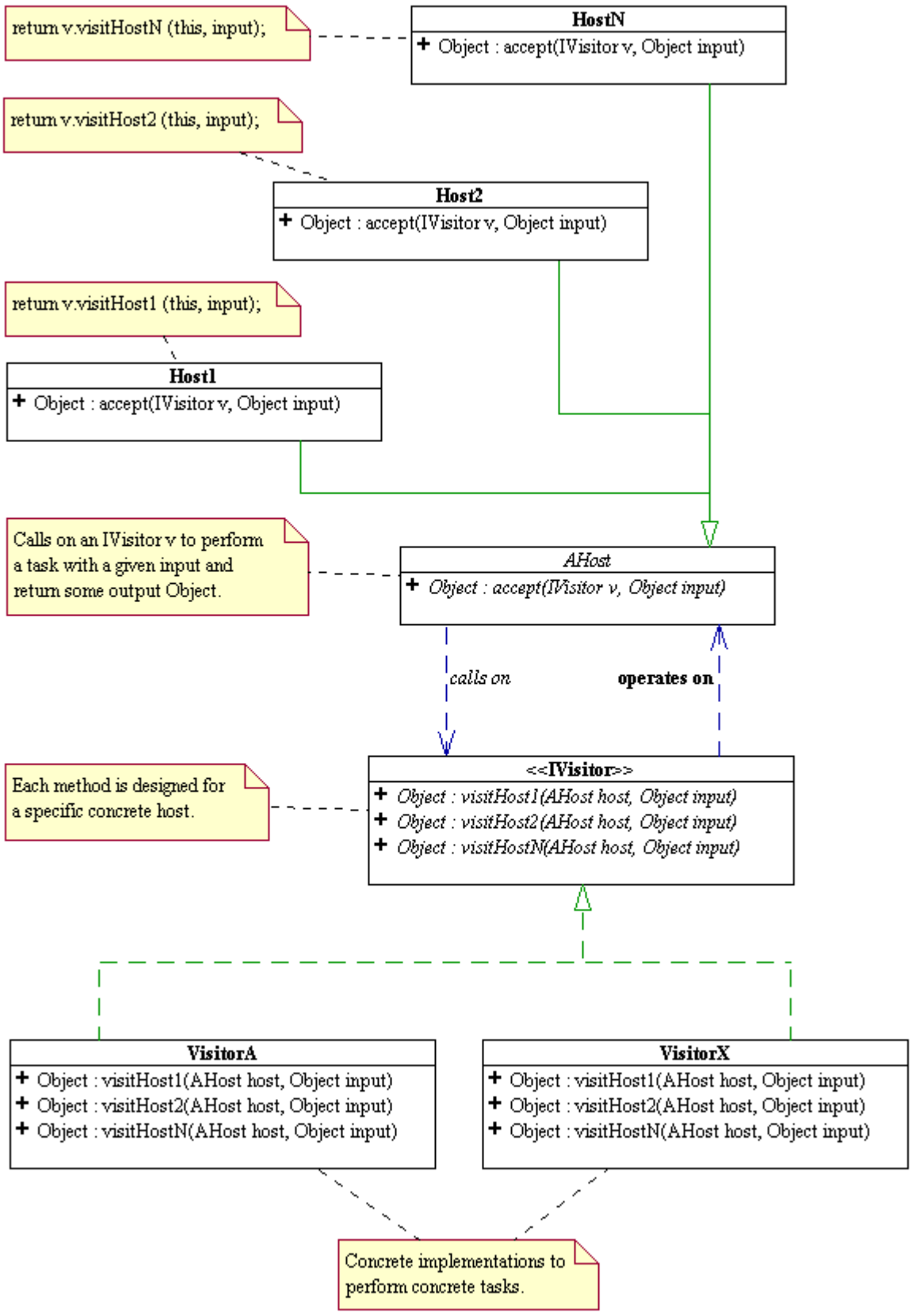
Limitations of the Interpreter Design Pattern

Up until now, each time we want to compute something new, we apply *the interpreter pattern* add appropriate methods to IntList and implement those methods in EmptyIntList and ConsIntList. This process of extending the capability of the list structure is error-prone at best and cannot be carried out if one does not own the source code for this structure. Any method added to the system can access the private fields of EmptyIntList and ConsIntList and modify them at will. In particular, the code can change _first and _rest of ConsIntList breaking the invariant immutable property the system is supposed to represent. The system so designed is inherently fragile, cumbersome, rigid, and limited. We end up with a forever changing IntList that includes a multitude of unrelated methods.

These design flaws come of the lack of delineation between the intrinsic and primitive behavior of the structure itself and the more complex behavior needed for a specific application. The failure to decouple primitive and non-primitive operations also causes reusability and extensibility problems. The weakness in bundling a data structure with a predefined set of operations is that it presents a static non-extensible interface to the client that cannot handle unforeseen future requirements. Reusability and extensibility are more than just aesthetic issues; in the real world, they are driven by powerful practical and economic considerations. Computer science students should be conditioned to design code with the knowledge that it will be modified many times. In particular is the need for the ability to add features *after* the software has been delivered. Therefore one must seek to *decouple* the data structures from the algorithms (or operations) that manipulate it. The Visitor Design Pattern is an object-oriented approach to address this issue.

The Visitor Design Pattern

The visitor pattern is a pattern for communication and collaboration between two union patterns: a "host" union and a "visitor" union. An abstract visitor is usually defined as an interface or an abstract class in Java. It has a separate method for each of the concrete variant of the host union. The abstract host has a method (called the "hook") to "accept" a visitor and leaves it up to each of its concrete variants to call the appropriate visitor method. This "decoupling" of the host's structural behaviors from the extrinsic algorithms on the host permits the addition of infinitely many external algorithms without changing any of the host union code. This extensibility only works if the taxonomy of the host union is stable and does not change. If we have to modify the host union, then we will have to modify ALL visitors as well! The following is diagram showing the overall architecture of the visitor design pattern. This type of diagram is called the UML (Unified Modeling Language) class diagram.



Applied to the list structure, the visitor pattern takes the following coding pattern.

```
/**
 * Abstract list structure.
 */
abstract class IntList {
    abstract Object accept(IntListVisitor v);

    ConsIntList cons(int n) {
        return new ConsIntList(n, this);
    }
}

/**
 * Concrete empty list structure containing nothing.
 */
class EmptyIntList extends IntList {
    Object accept(IntListVisitor v) {
        return v.forEmptyIntList(this);
    }
}

/**
 * Concrete non-empty list structure containing an int, called first,
 * and a rest, which is a list structure.
 */
class ConsIntList extends IntList {
    int first;
    IntList rest;

    Object accept(IntListVisitor v) {
        return v.forConsIntList(this);
    }
}

/**
 * Abstract operation on IntList.
 */
interface IntListVisitor {
    Object forEmptyIntList(EmptyIntList host);
    Object forConsIntList(ConsIntList host);
}
```

Here the host structure is IntList and its concrete subclasses, and the visitor is any extrinsic operation to be performed on the IntList host. Instead of adding methods to IntList to perform additional operations, we write visitors.

NOTE: Visitors that take no arguments should be written as Singletons.
We now translate code we wrote using the interpreter pattern on IntList in previous labs using visitors instead.

List Visitor Exercises

1. Write a visitor called Length to compute the length of an IntList.
    a. Write a non-tail recursive version (using direct recursion).
    b. Write a tail-recursive version using a helper visitor (in place of a helper method).
2. Write a visitor called `ProdNums` that returns the product of the number in the list, using a tail recursive helper visitor.
3. Write a visitor called `Reverse` that reverses the list using a tail-recursive helper visitor.
4. Write a visitor called `ListString` that uses as tail recursive visitor as done in the method listString.
5. Write a visitor called `MakePalindrome` that returns a list consisting of the input list and its mirror around the last element, using a (non tail-recursive) helper with an accumulator. For example, `(1, 2, 3).accept(MakePalindrome.ONLY)` returns the list `(1, 2, 3, 2, 1)` .
