A JML Specification of the Design Pattern "Visitor"*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz, Austria
Wolfgang.Schreiner@risc.jku.at

September 17, 2010

Abstract

We describe a generic Java framework that implements the software design pattern "visitor" and that is formally specified in the Java Modeling Language (JML). In addition to the information provided by a typical UML specification of the pattern, the JML specification describes the the sequence of visited objects in the order in which they are visited. A visitor may then specify its concrete behavior with respect to this sequence.

Contents

*Supported by the Austrian Academic Exchange Service (ÖAD) under the contract HU 14/2009.

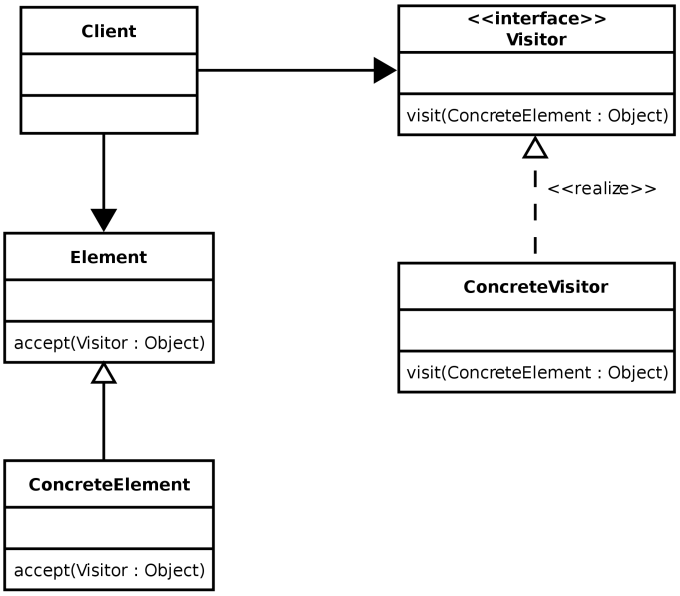

Figure 1: Visitor Pattern in UML (from Wikipedia)

1 Introduction

Like most software design patterns [2], the pattern “Visitor” is typically documented by a UML diagram (see Figure 1) accompanied by an extensive verbal description that explains the core idea, gives examples, indicates its applicability, outlines possible use cases, and so on. For example, the core description of Wikipedia on the topic “Visitor pattern” is as follows:

> In object-oriented programming and software engineering, the visitor design pattern is a way of separating an algorithm from an object structure it operates on. ...
>
> A user object receives a pointer to another object which implements an algorithm. The first is designated ’element class’ and the latter ’the visitor class’. The idea is to use a structure of element classes, each of which has an `accept()` method taking a visitor object for an argument. visitor is an interface having a `visit()` method for each element class. The `accept()` method of an element class calls back the `visit()` method for its class. Separate concrete visitor classes can then be written to perform some particular operations, by implementing these operations in their respective `visit()` methods. ...
>
> The following example is in the Java programming language, and shows how the contents of a tree of nodes (in this case describing the components of a car) can be printed. Instead of creating "print" methods for each subclass (Wheel, Engine, Body, and Car), a single class (CarElementPrintVisitor) performs the required printing action. Because different subclasses require slightly different actions to print properly,

> CarElementDoVisitor dispatches actions based on the class of the argument passed to it. ...

While in [2] the description is much more extensive and covers a dozen pages or so, the basic style is the same: UML diagrams and informal verbal descriptions aided by examples and code snippets. The only formally precise definition is represented by the UML diagrams which however typically only cover the static relationship between the classes/objects that make up the pattern. For instance, the diagram in Figure 1 says that an object of type `ConcreteElement` has a method `accept` and an object of type `ConcreteVisitor` has a method `visit`; the exact relationship between the methods is not clear. In particular, if the object is composed of subobjects, it is not a priori clear whether `accept` or `visit` is in charge of iterating over the object structure. The description in Wikipedia leaves this question essentially open:

> The visitor pattern also specifies how iteration occurs over the object structure. In the simplest version, where each algorithm needs to iterate in the same way, the `accept()` method of a container element, in addition to calling back the `visit()` method of the visitor, also passes the visitor object to the `accept()` method of all its constituent child elements.

"Visitor" is considered as a "behavioral" design pattern, i.e. it describes a particular mode of interaction, in this case the interaction between an object and an algorithm operating on this object. Analogous to our previous investigation of the structural design pattern "Proxy" [5], our goal is to make also the dynamic relationship between the methods formally precise; for this purpose we encode the pattern in a reusable Java framework which we formally specify with the Java Modeling Language (JML), a behavioral interface specification language for Java [4].

The remainder of this paper is structured as follows: In Section 2, we sketch the design of the framework. In Section 3, we discuss the JML specification of the framework. In Section 4 we develop and specify a concrete visitor on the basis of the generic framework. In Section 5, we summarize our experience and present our conclusions. Appendix A presents the full JML-annotated Java source code.

2 A Generic Visitor Framework in Java

Before developing a JML specification for the "Visitor" pattern, we first have to decide in which way to formulate the pattern in concrete Java code. One option is to develop a concrete example application of the pattern and to specify this application; however, then it is difficult to differentiate between those features that are fundamental to the pattern and those features that are peculiar for the particular use of the pattern. On the other side, we cannot remain completely abstract because a JML specification depends on concrete interfaces and classes.

As a consequence, we fix the interface of a visitor as

```
public interface Visitor
{
  public void visit(Element elem);
  public boolean check(Element elem);
```

```
  public void reset();
}
```

Here the method `visit()` represents a generic method that can be applied to any type of element (which must implement the interface `Element`); this is actually different from the original intention of the visitor pattern where the interface is intended to specify multiple `visit()` methods, one for *every* type of concrete element (because every concrete object is intended to call its own `visit()` method). In contrast, in our framework all concrete objects call the same generic `visit()` method, which must according to the concrete type of its argument `elem` dispatch the concrete `visit` method for this type.

We choose this deviation from the original intention of the visitor pattern because of the reason stated at the beginning of this section: we want to specify a single generic framework, which is only possible if we have a single interface that is independent of the concrete element classes. The price of this solution is, however, the additional runtime overhead for dispatching the actual `visit()` method of the concrete element class in the implementation of the visitor (see below). If this overhead is undesired, a different kind of `Visitor` interface has to be created for each concrete collection of element types; each `visit()` method in this interface has then to receive the same specification as the generic method illustrated above.

The methods `check()` and `reset()` are actually not part of the original visitor pattern (and can be also omitted); after a call of `visit()`, `check()` allows to verify from information collected in the visitor object whether all required (sub)objects have been visited in the right order; the method `reset()` clears this information.

As for elements, we provide an interface

```
public interface Element
{
  public void accept(Visitor visitor);
  public int numberOfChildren();
  public Element getChild(int n);
}
```

Here the method `accept()` accepts the visitor as described in the pattern. However, we also require two additional methods: the method `numberOfChildren()` returns the number of those subobjects of which the current object is composed and that have to be correspondingly traversed by a visitor; the method `getObject()` allows to determine each subobject.

We introduce `numberOfChildren()` and `getObject()` because in our framework it is the `visit()` method that is in charge of iterating over an object structure: each visitor thus has the chance to perform some action before and after the traversal of each individual child. To make `accept()` responsible for the traversal (as described in the Wikipedia article cited in the introduction) would make the framework too restrictive for practical purposes.

For each of the two interfaces, we introduce an abstract class as as basis for a suitable implementation of the interface.

```
public abstract class VisitorBase implements Visitor
{
  private Element[] visited;
```

```
protected VisitorBase()
{
  this.visited = new Element[0];
}

public void visit(Element elem)
{
  add(elem);
  childrenAccept(elem);
}

protected final void add(Element elem)
{
  int n = visited.length;
  Element[] visited0 = new Element[n+1];
  for (int i=0; i<n; i++) visited0[i] = visited[i];
  visited0[n] = elem;
  visited = visited0;
}

protected final void childrenAccept(Element elem)
{
  int n = elem.numberOfChildren();
  for (int i=0; i<n; i++)
    elem.getChild(i).accept(this);
}

public boolean check(Element elem)
{
  return (visited(0, elem) == visited.length);
}

private int visited(int pos, Element elem)
{
  if (pos >= visited.length) return -1;
  if (visited[pos] != elem) return -1;
  int pos0 = pos+1;
  int n = elem.numberOfChildren();
  for (int i=0; i<n; i++)
  {
    int pos1 = visited(pos0, elem.getChild(i));
    if (pos1 == -1) return -1;
    pos0 = pos1;
  }
  return pos0;
}

public void reset()
{
```

```
    this.visited = new Element[0];
  }
}
```

The class `VisitorBase` introduces a private variable `visited` that collects the nodes visited so far in the order of their visit; for this purpose the visitor may use a method `add`. The method `childrenAccept` lets the subobjects of the current object accept the visitor. The default implementation of `visit()` shows a minimal legal implementation. The method `check()` verifies with the help of an auxiliary method `visited()` whether the object hierarchy has been visited in the correct order; `visited`() verifies whether `elem` has been recorded in the array `visited` starting at position `pos` and returns the first position that does not any more belong to this visit (-1, if an error is observed).

```
public abstract class ElementBase implements Element
{
  private Element[] children;

  protected ElementBase(Element[] children)
  {
    this.children = children;
  }

  public final int numberOfChildren()
  {
    return children.length;
  }

  public final Element getChild(int n)
  {
    return children[n];
  }

  public final void accept(Visitor visitor)
  {
    visitor.visit(this); //@ nowarn; // checking takes too long
  }
}
```

The class `ElementBase` provides the base for an implementation of a composed element with an implementation of `accept()`.

3 A JML Specification of the Framework

The core problem of the JML specification is to describe that a visitor has visited an element and all of its subelements in the correct order.

We attempt a solution to this problem by introducing in interface `Visitor` a JML model variable (specification-only mathematical variable) `visitedM` that denotes the sequence of elements visited so far:

```
public interface Visitor
{
  /*@ instance non_null public model Element[] visitedM; @*/

  /*@ public normal_behavior
    @ requires elem != null && elem.isTree();
    @ assignable \everything;
    @ ensures elem.isFlattenedTo(\old(visitedM), visitedM);
    @*/
  public void visit(Element elem);

  /*@ public normal_behavior
    @ requires elem != null;
    @ ensures \result = elem.isFlattened(visitedM, 0, visitedM.length);
    @ pure @*/ public boolean check(Element elem);

  /*@ protected normal_behavior
    @ assignable visitedM;
    @ ensures visitedM.length == 0;
    @*/
  public void reset();
}
```

The actual specification is delegated to two predicates: `isFlattenedTo()` is true, if the new value of `visitedM` after the call of `visit()` is derived from the old value by flattening the object hierarchy rooted in `elem`; `isFlattened` checks whether `visitedM` contains in its whole range the flattened version of the object hierarchy rooted in `elem`. The predicates are introduced as model methods in the interface `Element`.

```
public interface Element
{
  //@ public non_null instance model Element[] childrenM;

  // the element hierarchy rooted in this element represents a tree
  /*@ public invariant isTree(); @*/

  /*@ public normal_behavior
    @ requires visitor != null;
    @ assignable \everything;
    @ ensures isFlattenedTo(\old(visitor.visitedM), visitor.visitedM);
    @*/
  public void accept(Visitor visitor);

  /*@ public normal_behavior
    @ ensures \result == childrenM.length;
    @*/
  /*@ pure @*/ public int numberOfChildren();

  /*@ public normal_behavior
    @ requires 0 <= n && n < childrenM.length;
```

```
    @ ensures \result == childrenM[n];
    @ ensures_redundantly \result != null && \result.isTree();
    @*/
  /*@ pure @*/ public Element getChild(int n);

  // model predicates and functions
  ...
}
```

In interface `Element` a model variable `childrenM` denotes the subobjects of this object; the invariant `isTree()` (the predicates is later introduced as a model function) denotes that the object hierarchy forms a tree (in particular, no cycles are allowed). The method `accept` specifies that the element hierarchy rooted in this element is added to `visitedM`; the methods `numberOfChildren()` and `getChild()` specify that they derive their values from `childrenM`.

The predicates used in above specifications are introduced as model methods in `Element`:

```
  /*@ public normal_behavior
    @ ensures \result == validChildren(childrenM);
    @ public pure model boolean isTree();
    @*/

  /*@ public normal_behavior
    @ ensures \result == true <==>
    @   disjointTrees(children) &&
    @   (\forall int i; 0 <= i && i < children.length;
    @      !children[i].isReachable(this));
    @ public pure model boolean validChildren(Element[] children);
    @*/

  /*@ protected normal_behavior
    @ requires elements != null;
    @ ensures \result == true <==>
    @  (0 <= begin && begin < end && end <= elements.length &&
    @    elements[begin] == this &&
    @    childrenAreFlattened(elements, begin+1, end));
    @ model pure boolean isFlattened(Element[] elements, int begin, int end);
    @*/

  /*@ public normal_behavior
    @ requires elems0!= null && elems1 != null;
    @ ensures \result == true <==>
    @   elems1.length == elems0.length+size() &&
    @   isPrefix(elems0, elems1) &&
    @   isFlattened(elems0, elems0.length, elems1.length);
    @ model pure boolean isFlattenedTo(Element[] elems0, Element[] elems1);
    @*/

   ... // more model predicates and functions
```

A full listing of all model functions and their specifications is given in the appendix; they

essentially introduce a theory of trees and their pre-order flattening into a linear sequence of nodes (with some consequences that may become useful in static checking and verification).

The class `VisitorBase` may now be extended by a specification of private invariants and method behaviors:

```
public abstract class VisitorBase implements Visitor
{
  // the visited nodes
  /*@ non_null @*/ private Element[] visited; /*@ in visitedM; @*/

  /*@ private represents visitedM = visited; @*/
  /*@ private invariant
    @   (\forall int i; 0 <= i && i < visited.length;
    @     visited[i] != null && visited[i].isTree());
    @*/

  /*@ protected normal_behavior
    @ assignable visitedM;
    @ ensures visitedM.length == 0;
    @ also private normal_behavior
    @ assignable visitedM;
    @ ensures visited.length == 0;
    @*/
  protected VisitorBase() { ... }

  public void reset() { this.visited = new Element[0]; }

  /*@ protected normal_behavior
    @ requires elem != null;
    @ assignable visitedM;
    @ ensures elem.isAddedTo(\old(visitedM), visitedM);
    @*/
  protected final void add(Element elem) { ... }

  /*@ protected normal_behavior
    @ requires elem != null;
    @ assignable \everything;
    @ ensures elem.childrenAreFlattenedTo(\old(visitedM), visitedM);
    @*/
  protected final void childrenAccept(Element elem) { ... }

  public boolean check(Element elem) { ... }

  /*@ private normal_behavior
    @ requires elem != null && 0 <= pos && pos <= visited.length;
    @ assignable \nothing;
    @ ensures \result != -1 ==>
    @   pos < \result && \result <= visited.length &&
    @   elem.isFlattened(visited, pos, \result);
    @ ensures \result == -1 ==>
```

```
    @   !(\exists int result0; pos < result0 && result0 <= visited.length;
    @      elem.isFlattened(visited, pos, result0));
    @*/
  /*@ pure helper @*/ private int visited(int pos, Element elem) { ... }
}
```

To validate the specification, we have applied the extended static checker ESC/Java2; except for the postconditions of `childrenAccept()` and `visited()` it does not complain (however, an additional invariant has to be introduced, see the full listing in the appendix).

Also the class `ElementBase` can be annotated with private behavior specifications:

```
public abstract class ElementBase implements Element
{
  /*@ non_null @*/ private Element[] children; //@ in childrenM;
  /*@ private represents childrenM = children; @*/

  /*@ private normal_behavior
    @ requires disjointTrees(children);
    @ assignable this.children;
    @ ensures this.children == children;
    @ also protected normal_behavior
    @ requires disjointTrees(children);
    @ assignable childrenM;
    @ ensures childrenM == children;
    @*/
  protected ElementBase(Element[] children) { ... }

  /*@ also private normal_behavior
    @ ensures \result == children.length;
    @*/
  public final int numberOfChildren() { ... }

  /*@ also private normal_behavior
    @ requires 0 <= n && n < children.length;
    @ ensures \result == children[n];
    @*/
  public final Element getChild(int n) { ... }

  public final void accept(Visitor visitor) { ... }
}
```

ESC/Java2 cannot check the preservation of the invariant of the constructor and of the postcondition of `accept` in a reasonable amount of time; otherwise it does not complain.

4 A Sample Use of the Framework

We validate the use of the framework by a small application that introduces binary trees labeled with integer numbers and a visitor that computes the sum of the node labels.

The binary tree is introduced by the following classes:

```
public abstract class Node extends ElementBase
{
  final static int INNER = 1;
  final static int LEAF = 2;

  public final int tag;
  /*@ public invariant
    @ (tag == INNER ==> this instanceof Inner) &&
    @ (tag == LEAF  ==> this instanceof Leaf);
    @*/

  /*@ requires disjointTrees(nodes);
    @ assignable childrenM, this.tag;
    @ ensures childrenM == nodes && this.tag == tag;
    @*/
  protected Node(Node[] nodes, int tag)
  {
    super(nodes);
    this.tag = tag;
  } //@ nowarn Invariant; // checking takes too long
}

public class Leaf extends Node
{
  //@ invariant numberOfChildren() == 0;

  public Leaf()
  {
    super(new Node[0], LEAF);
  }
}

public class Inner extends Node
{
  // disable checking with option -NoCheck since checker seems to loop

  private int label;

  //@ invariant numberOfChildren() == 2;

  /*@ private normal_behavior
    @ requires child1 != null && child2 != null;
    @ requires child1.isTree() && child2.isTree() && child1.disjoint(child2);
    @ assignable childrenM, this.tag, this.label;
    @ ensures childrenM != null && childrenM.length == 2;
    @ ensures childrenM[0] == child1 && childrenM[1] == child2;
    @ ensures this.tag == INNER && this.label == label;
    @*/
  public Inner(int label, Node child1, Node child2)
```

```
  {
    super(new Node[] { child1, child2 }, INNER);
    this.label = label;
  }

  /*@ private normal_behavior
    @ ensures \result == label;
    @*/
  /*@ pure @*/ public int getLabel()
  {
    return label;
  }
}
```

Every tree node is either an inner node of type `Inner` or a leaf node of type `Leaf`. To simplify the dispatch of the concrete `visit()` methods, `Inner` nodes are tagged with a numerical tag `INNER` and `Leaf` nodes with the tag `LEAF`. An invariant specifies the corresponding relationship.

Tree visitors can be based on an abstract class `NodeVisitorBase`:

```
public abstract class NodeVisitorBase extends VisitorBase
{
  /*@ public normal_behavior
    @ assignable visitedM;
    @ ensures visitedM.length == 0;
    @*/
  public NodeVisitorBase()
  {
    super();
  }

  public final void visit(Element elem)
  {
    if (!(elem instanceof Node))
    {
      super.visit(elem);
      return;
    }
    add(elem);
    switch(((Node)elem).tag)
    {                                                  // else ESC/Java2 runs forever
    case Node.INNER: visit((Inner)elem); break; //@ nowarn;
    case Node.LEAF:  visit((Leaf)elem); break;  //@ nowarn;
    default: childrenAccept(elem); break;
    }
  }

  /*@ public normal_behavior
    @ requires elem != null && elem.isTree();
    @ assignable \everything;
    @ ensures elem.childrenAreFlattenedTo(\old(visitedM), visitedM);
```

```
    @*/
  public void visit(Inner elem) { childrenAccept(elem); };

  /*@ public normal_behavior
    @ requires elem != null && elem.isTree();
    @ assignable \everything;
    @ ensures elem.childrenAreFlattenedTo(\old(visitedM), visitedM);
    @*/
  public void visit(Leaf elem) { childrenAccept(elem); };
}
```

The method `visit` in this class implements the method specified in the `Visitor` interface (overriding the default implementation in `VisitorBase`). It adds the current node to the list of visited nodes and then performs the dispatch to the `visit` method of the concrete class (which only has to take care to add the child elements to the list).

Based on this class, the concrete visitor can be defined as follows:

```
public final class NodeAdder extends NodeVisitorBase
{
  private int sum; //@ in visitedM;

  /*@ public normal_behavior
    @ assignable visitedM;
    @ ensures visitedM.length == 0 && getSum() == 0;
    @*/
  public NodeAdder() { super(); sum = 0; }

  /*@ also public normal_behavior
    @ assignable visitedM;
    @ ensures getSum() == 0;
    @*/
  public void reset() { super.reset(); sum = 0; }

  /*@ public normal_behavior
    @ ensures \result == getSum();
    @ also private normal_behavior
    @ ensures \result == sum;
    @*/
  /*@ pure @*/ public int getSum() { return sum; }

  /*@ also public normal_behavior
    @ requires node != null && node.isTree();
    @ requires hasSum(visitedM.length-1);
    @ assignable \everything;
    @ ensures hasSum(visitedM.length);
    @*/
  public void visit(Inner node)
  {
    sum = sum + node.getLabel();
    childrenAccept(node); //@ nowarn;
```

```
} //@ nowarn Post; // postcondition can't be established

/*@ public normal_behavior
  @ requires 0 <= pos && pos < visitedM.length;
  @ ensures \result == true <==>
  @   getSum() ==
  @     (\sum int i; 0 <= i && i < pos;
  @        (visitedM[i] instanceof Inner) ?
  @          ((Inner)visitedM[i]).getLabel() : 0);
  @ public pure model boolean hasSum(int pos);
  @*/
```

The `visit()` method for leaf nodes need not be defined (since the default implementation can be inherited from `NodeVisitorBase`); the `visit()` method for inner nodes traverses the children and adds the value of the current label to the sum. The specification claims that after a call of this method, `getSum()` returns the sum of all values of all visited inner nodes, provided that before the call it returns the sum of all values of all visited nodes except the currently visited one (as can be seen from the definition of method `visit()` in class `NodeVisitorBase`, the method `visit()` in class `NodeAdder` is called in a state when the current element has already been added but the sum value has not been updated yet). Since the constructor `NodeAdder()` establishes that no nodes have been visited yet and `getSum()` returns zero, we can deduce that by a finite number of `visit()` calls, the property remains invariant.

Nevertheless the property must *not* be stated as an invariant of the class, since it is violated by the inherited method `add()` which adds an element to the sequence of visited elements. Thus the property does not hold at all observable states of the visitor object (after the call of any method of the object) but only after the call of `visit()`. The very existence of the `add()` method thus generally prevents the strengthening of the invariants of visitor objects; all additional information must be attached to the constructor of the object and to its `visit()` methods as explicit pre- and postconditions.

ESC/Java2 can be used to validate these classes with the following exceptions:

- In `Node`, the checking of the invariant at the end of the constructor does not terminate in a moderate amount of time (it is therefore switched off).
- The checking of `Inner` does not terminate in a moderate amount of time.
- In `NodeVisitorBase`, the checking of the calls of the `visit()` methods for the concrete element types in the generic `visit()` methods takes a long time and is switched off (the check has however succeeded with another slightly different specification of the generic framework).
- In `NodeAdder`, the correctness of the postcondition of `visit` for inner nodes cannot be established; furthermore the checking of the call of `childrenAccept()` has to be turned off since it takes too long.

Ultimately, the visitor can be applied as demonstrated by the following program:

```
public class Main
{
  public static void main(String[] args)
  {
    Inner tree = new Inner(1, new Inner(2, new Leaf(),
        new Inner(3, new Leaf(), new Leaf())),
        new Inner(4, new Leaf(), new Leaf()));
    NodeAdder adder = new NodeAdder();
    tree.accept(adder);
    System.out.println(adder.getSum());
    System.out.println(adder.check(tree));
  }
}
```

The output of the program is

```
10
true
```

i.e. the visitor has computed the expected result which is validated by a runtime check.

5 Conclusions

The JML specification of the behavioral design pattern "Visitor" essentially relates a call of a visitor to the sequence of elements encountered by the visitor. For this purpose, we have introduced in the specification an additional model variable that denotes the sequence and have further specified all methods with the help of this variable. A concrete visitor may then describe its specific behavior in relationship to this sequence.

As for the previously specified pattern "Proxy", one can consider this technique as the special case of a "history variable" [3] that records previous states and thus captures the temporal behavior of a program in a data structure. It is, however, unclear whether this is really the most appropriate way to formulate the temporal behavior of the design pattern. As an alternative, one may also consider a temporal specification of program events in e.g. LTL [6] and use e.g. a runtime verification framework such as MOP [1] to validate the specification.

In any case, since already two different design patterns have yielded similar techniques, these techniques may possibly serve as the core of a general strategy for the specification of the dynamic behavior of design patterns. Further work will help to substantiate or refute this claim.

References

[1] Feng Chen and Grigore Roşu. MOP: An Efficient and Generic Runtime Verification Framework. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 569–588, New York, NY, USA, 2007. ACM.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1995.

[3] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison Wesley, 2002.

[4] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06y, Department of Computer Science, Iowa State University, June 2004. See www.jmlspecs.org.

[5] Wolfgang Schreiner. A JML Specification of the Design Pattern "Visitor". Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University, Linz, Austria, April 2009.

[6] Kerry Trentelman and Marieke Huisman. Extending JML Specifications with Temporal Logic. In *Algebraic Methodology And Software Technology (AMAST'02), St. Gilles les Bains, Reunion Island, France, September 9–13*, volume 2422 of *Lecture Notes in Computing Series*, pages 334—348. Springer, 2002.

A The JML-annotated Java Code

A.1 Visitor Interfaces

```
package patterns.visitor;

/*************************************************************************
 * Interface to a visitor.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 *************************************************************************/
public interface Visitor
{
  /*@ instance non_null public model Element[] visitedM; @*/

  /*************************************************************************
   * Visit element.
   * @param elem the element to be visited.
   *************************************************************************/
  /*@ public normal_behavior
    @ requires elem != null && elem.isTree();
    @ assignable \everything;
    @ ensures elem.isFlattenedTo(\old(visitedM), visitedM);
    @*/
  public void visit(Element elem);

  /*************************************************************************
   * Check whether elem has been visited in top-down order.
   * @return true if elem and its children have been visited in top-down order.
   *************************************************************************/
  /*@ public normal_behavior
    @ requires elem != null;
    @ ensures \result == elem.isFlattened(visitedM, 0, visitedM.length);
    @ pure @*/ public boolean check(Element elem);
```

```
  /*************************************************************************
   * Reset the list of visited nodes;
   * must be called before a new call of check().
   *************************************************************************/
  /*@ protected normal_behavior
    @ assignable visitedM;
    @ ensures visitedM.length == 0;
    @*/
  public void reset();
}

package patterns.visitor;

/*************************************************************************
 * Interface to an element to be visited.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 *************************************************************************/
public interface Element
{
  //@ public non_null instance model Element[] childrenM;

  // the element hierarchy rooted in this element represents a tree
  /*@ public invariant isTree(); @*/

  /*************************************************************************
   * Get the number of children of this element.
   * @return the number of children of this element.
   *************************************************************************/
  /*@ public normal_behavior
    @ ensures \result == childrenM.length;
    @*/
  /*@ pure @*/ public int numberOfChildren();

  /*************************************************************************
   * Get the n-th child of this element.
   * @param n the number of the child (0 <= n < numberOfChildren())
   *************************************************************************/
  /*@ public normal_behavior
    @ requires 0 <= n && n < childrenM.length;
    @ ensures \result == childrenM[n];
    @ ensures_redundantly \result != null && \result.isTree();
    @*/
  /*@ pure @*/ public Element getChild(int n);

  /*************************************************************************
   * Accept a visitor.
   * @param visitor the visitor accepted by the element.
   *************************************************************************/
  /*@ public normal_behavior
    @ requires visitor != null;
    @ assignable \everything;
    @ ensures isFlattenedTo(\old(visitor.visitedM), visitor.visitedM);
```

```
  @*/
public void accept(Visitor visitor);

/*************************************************************************
 *
 * Graph-theoretic notions below.
 *
 *************************************************************************/

/*************************************************************************
 * True if children form disjoint trees that do not reach the root
 *************************************************************************/
/*@ public normal_behavior
  @ ensures \result == validChildren(childrenM);
  @
  @ // consequently there is a unique path to every node in the tree
  @ ensures_redundantly \result == true <==>
  @   (\forall Element elem; elem != null && isReachable(elem);
  @     (\forall Element[] path, path0;
  @       path != null && reaches(elem, path) &&
  @       path0 != null && reaches(elem, path0);
  @          path.length == path0.length && isPrefix(path, path0)));
  @
  @ // consequently there are no cycles in the tree
  @ ensures_redundantly !hasCycles();
  @
  @ public pure model boolean isTree();
  @*/

/*************************************************************************
 * True if these are valid children for a hierarchy rooted in this element.
 *************************************************************************/
/*@ public normal_behavior
  @ ensures \result == true <==>
  @   disjointTrees(children) &&
  @   (\forall int i; 0 <= i && i < children.length;
  @      !children[i].isReachable(this));
  @ public pure model boolean validChildren(Element[] children);
  @*/

/*************************************************************************
 * True if these nodes form disjoint trees.
 *************************************************************************/
/*@ public normal_behavior
  @ ensures \result == true <==>
  @   elems != null && \nonnullelements(elems) &&
  @   (\forall int i; 0 <= i && i < elems.length; elems[i].isTree()) &&
  @   (\forall int i, j;
  @     0 <= i && i < elems.length &&
  @     0 <= j && j < elems.length && i != j;
  @       elems[i].disjoint(elems[j]));
  @ public static pure model boolean disjointTrees(Element[] elems);
  @*/
```

```
/*************************************************************************
 * True if hierachy denoted by element is disjoint from this hierarchy
 *************************************************************************/
/*@ public normal_behavior
  @ ensures \result == true <==>
  @   elem != null &&
  @   !(\exists Element elem0; elem0 != null;
  @     this.isReachable(elem0) && elem.isReachable(elem0));
  @ public pure model boolean disjoint(Element elem);
  @*/

/*************************************************************************
 * True if this element is parent of elem.
 *************************************************************************/
/*@ public normal_behavior
  @ requires elem != null;
  @ ensures \result == true <==>
  @  (\exists int i; 0 <= i && i < numberOfChildren(); getChild(i) == elem);
  @ public pure model boolean isParentOf(Element elem);
  @*/

/*************************************************************************
 * True if this element is ancestor of elem.
 *************************************************************************/
/*@ public normal_behavior
  @ requires elem != null;
  @ ensures \result == true <==>
  @   isParentOf(elem) ||
  @   (\exists Element elem0; elem0 != null;
  @     isParentOf(elem0) && elem0.isAncestorOf(elem));
  @ model pure boolean isAncestorOf(Element elem);
  @*/

/*************************************************************************
 * True if elem is reachable from this element.
 *************************************************************************/
/*@ public normal_behavior
  @ requires elem != null;
  @ ensures \result == true <==>
  @   this == elem || isAncestorOf(elem);
  @ model pure boolean isReachable(Element elem);
  @*/

/*************************************************************************
 * This element reaches elem via the denoted path.
 *************************************************************************/
/*@ public normal_behavior
  @ requires elem != null && path != null;
  @ ensures \result == true <==>
  @  path.length > 0 && path[0] == this && path[path.length-1] == elem &&
  @  (\forall int i; 0 <= i && i < path.length-1;
  @    path[i].isParentOf(path[i+1]));
  @ public pure model boolean reaches(Element elem, Element[] path);
  @*/
```

```
/*************************************************************************
 * An element reachable by this element is its own ancestor.
 *************************************************************************/
/*@ public normal_behavior
  @ ensures \result == true <==>
  @   (\exists Element elem; elem != null && isReachable(elem);
  @     elem.isAncestorOf(elem));
  @ public pure model boolean hasCycles();
  @*/

/*************************************************************************
 * True if elems1 is derived from elems0 by adding this element.
 *************************************************************************/
/*@ protected normal_behavior
  @ requires elems0 != null && elems1 != null;
  @ ensures (\result == true) <==>
  @   elems1.length == elems0.length+1 &&
  @   isPrefix(elems0, elems1) &&
  @   elems1[elems0.length] == this;
  @ model pure boolean
  @ isAddedTo(Element[] elems0, Element[] elems1);
  @*/

/*************************************************************************
 * True if the hierarchy rooted in this element is flattened into the array
 * elements from position begin (inclusive) to position end (exclusive)?
 *************************************************************************/
/*@ protected normal_behavior
  @ requires elements != null;
  @
  @ // recursive definition of predicate
  @ ensures \result == true <==>
  @  (0 <= begin && begin < end && end <= elements.length &&
  @    elements[begin] == this &&
  @    childrenAreFlattened(elements, begin+1, end));
  @
  @ // consequently, the size of the region is the number of nodes
  @ ensures_redundantly \result == true <==> end-begin == size();
  @
  @ model pure boolean isFlattened(Element[] elements, int begin, int end);
  @*/

/*************************************************************************
 * True if elems1 is derived from elems0 by flattening this element.
 *************************************************************************/
/*@ public normal_behavior
  @ requires elems0!= null && elems1 != null;
  @ ensures \result == true <==>
  @   elems1.length == elems0.length+size() &&
  @   isPrefix(elems0, elems1) &&
  @   isFlattened(elems0, elems0.length, elems1.length);
  @ model pure boolean isFlattenedTo(Element[] elems0, Element[] elems1);
  @*/
```

```
/*************************************************************************
 * Are the children of this element flattened into the array
 * elements from position begin (inclusive) to position end (exclusive)?
 *************************************************************************/
/*@ protected normal_behavior
  @ requires elements != null;
  @
  @ // recursive definition of predicate
  @ ensures \result == true <==>
  @  (0 <= begin && begin <= end && end <= elements.length &&
  @    (\exists int[] pos; pos != null && pos.length == numberOfChildren()+1;
  @      pos[0] == begin && pos [numberOfChildren()] == end &&
  @        (\forall int i; 0 <= i && i < numberOfChildren();
  @          pos[i] < pos[i+1] &&
  @          getChild(i).isFlattened(elements, pos[i], pos[i+1]))));
  @
  @ // consequently, the size of the region is the number of childnodes
  @ ensures_redundantly \result == true <==> end-begin == childrenSize();
  @
  @ model pure boolean
  @ childrenAreFlattened(Element[] elements, int begin, int end);
  @*/

/*************************************************************************
 * True if elems1 is derived from elems0 by flattening the children
 * of this element.
 *************************************************************************/
/*@ public normal_behavior
  @ requires elems0 != null && elems1 != null;
  @ ensures \result == true <==>
  @   elems1.length == elems0.length + childrenSize() &&
  @   isPrefix(elems0, elems1) &&
  @   childrenAreFlattened(elems0, elems0.length, elems1.length);
  @ model pure boolean childrenAreFlattenedTo(Element[] elems0, Element[] elems1);
  @*/

/*************************************************************************
 * True if the two arrays are the same in the positions of the first array.
 *************************************************************************/
/*@ protected normal_behavior
  @ requires elems0 != null && elems1 != null;
  @ ensures (\result == true) <==>
  @   elems0.length <= elems1.length &&
  @   (\forall int i; 0 <= i && i < elems0.length; elems0[i] == elems1[i]);
  @ model pure static boolean
  @ isPrefix(Element[] elems0, Element[] elems1);
  @*/

/*************************************************************************
 * The number of all nodes in the hierarchy rooted in this element
 *************************************************************************/
/*@ public normal_behavior
  @ ensures \result == 1 + childrenSize();
```

```
    @ ensures_redundantly \result > 0;
    @ public pure model int size();
    @*/

  /*************************************************************************
   * The number of all nodes of the children of this element
   *************************************************************************/
  /*@ public normal_behavior
    @ ensures \result ==
    @   (\sum int i; 0 <= i && i < numberOfChildren(); getChild(i).size());
    @ ensures_redundantly \result >= 0;
    @ public pure model int childrenSize();
    @*/
}
```

A.2 Visitor Base Classes

```
package patterns.visitor;

/*************************************************************************
 * Base class of a visitor.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 *************************************************************************/
public abstract class VisitorBase implements Visitor
{
  // the visited nodes
  /*@ non_null @*/ private Element[] visited; /*@ in visitedM; @*/

  /*@ private represents visitedM = visited; @*/
  /*@ private invariant
    @   (\forall int i; 0 <= i && i < visited.length;
    @     visited[i] != null && visited[i].isTree());
    @*/

  // without introducing getVisited() and this invariant, ESC/Java2 runs forever
  /*@ protected invariant visitedM == getVisited(); @*/

  /*************************************************************************
   * Get the list of visited nodes; only needed for invariant above.
   *************************************************************************/
  /*@ private normal_behavior
    @ ensures \result == visited;
    @ ensures_redundantly
    @    (\forall int i; 0 <= i && i < \result.length;
    @       \result[i] != null && \result[i].isTree());
    @*/
  /*@ pure @*/ protected Element[] getVisited()
  {
    return visited;
  }

  /*************************************************************************
   * Create the visitor.
```

```
 ****************************************************************************/
/*@ protected normal_behavior
  @ assignable visitedM;
  @ ensures visitedM.length == 0;
  @ also private normal_behavior
  @ assignable visitedM;
  @ ensures visited.length == 0;
  @*/
protected VisitorBase()
{
  this.visited = new Element[0];
}

/****************************************************************************
 * Reset the list of visited nodes;
 * must be called before a new call of check().
 ****************************************************************************/
/*@ also private normal_behavior
  @ assignable visitedM;
  @ ensures visited.length == 0;
  @*/
public void reset()
{
  this.visited = new Element[0];
}

/****************************************************************************
 * Visit element.
 * @param elem the element to be visited.
 ****************************************************************************/
public void visit(Element elem)
{
  add(elem);
  childrenAccept(elem);
}

/****************************************************************************
 * Add element to list of visited nodes.
 * @param elem the element to be added.
 ****************************************************************************/
/*@ protected normal_behavior
  @ requires elem != null;
  @ assignable visitedM;
  @ ensures elem.isAddedTo(\old(visitedM), visitedM);
  @*/
protected final void add(Element elem)
{
  int n = visited.length;
  Element[] visited0 = new Element[n+1];
  for (int i=0; i<n; i++) visited0[i] = visited[i];
  visited0[n] = elem;
  visited = visited0;
}
```

```
/*************************************************************************
 * Let children of elem accept this visitor.
 * @param elem the element whose children are to accept this visitor.
 *************************************************************************/
/*@ protected normal_behavior
  @ requires elem != null;
  @ assignable \everything;
  @ ensures elem.childrenAreFlattenedTo(\old(visitedM), visitedM);
  @*/
protected final void childrenAccept(Element elem)
{
  int n = elem.numberOfChildren();
  for (int i=0; i<n; i++)
    elem.getChild(i).accept(this);
} //@ nowarn Post; // post-condition cannot be established

/*************************************************************************
 * Check whether elem has been visited in top-down order.
 * @return true if elem and its children have been visited in top-down order.
 *************************************************************************/
public boolean check(Element elem)
{
  return (visited(0, elem) == visited.length);
}

/*************************************************************************
 * Check whether children of elem have been traversed in top-down order.
 * @param pos the position of the first visited element.
 * @param elem the element whose children are to be checked.
 * @return the position of the first visited element after the children
 *         (-1, if the check failed).
 *************************************************************************/
/*@ private normal_behavior
  @ requires elem != null && 0 <= pos && pos <= visited.length;
  @ assignable \nothing;
  @ ensures \result != -1 ==>
  @   pos < \result && \result <= visited.length &&
  @   elem.isFlattened(visited, pos, \result);
  @ ensures \result == -1 ==>
  @   !(\exists int result0; pos < result0 && result0 <= visited.length;
  @      elem.isFlattened(visited, pos, result0));
  @*/
/*@ pure helper @*/ private int visited(int pos, Element elem)
{
  if (pos >= visited.length) return -1;
  if (visited[pos] != elem) return -1;
  int pos0 = pos+1;
  int n = elem.numberOfChildren();
  for (int i=0; i<n; i++)
  {
    int pos1 = visited(pos0, elem.getChild(i));
    if (pos1 == -1) return -1;
    pos0 = pos1;
  }
```

```
    return pos0;
  } //@ nowarn Post; // post-condition cannot be established
}

package patterns.visitor;

/**********************************************************************
 * Base class of an element to be visited.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 **********************************************************************/
public abstract class ElementBase implements Element
{
  // the children elements
  /*@ non_null @*/ private Element[] children; //@ in childrenM;
  /*@ private represents childrenM = children; @*/

  /**********************************************************************
   * ElementBase(children)
   * Create an element with the denoted children elements
   *
   * @param children the children elements (non-null)
   **********************************************************************/
  /*@ private normal_behavior
    @ requires disjointTrees(children);
    @ assignable this.children;
    @ ensures this.children == children;
    @ also protected normal_behavior
    @ requires disjointTrees(children);
    @ assignable childrenM;
    @ ensures childrenM == children;
    @*/
  protected ElementBase(Element[] children)
  {
    this.children = children;
  } //@ nowarn Invariant; // checking takes too long

  /**********************************************************************
   * n = numberOfChildren()
   * n is the number of children of this element.
   *
   * @return the number of children of this element.
   **********************************************************************/
  /*@ also private normal_behavior
    @ ensures \result == children.length;
    @*/
  public final int numberOfChildren()
  {
    return children.length;
  }

  /**********************************************************************
   * Get the n-th child of this element.
   * @param n the number of the child (0 <= n < number of children)
```

```
   ********************************************************************/
  /*@ also private normal_behavior
    @ requires 0 <= n && n < children.length;
    @ ensures \result == children[n];
    @*/
  public final Element getChild(int n)
  {
    return children[n];
  }

  /**********************************************************************
   * Accept a visitor.
   * @param visitor the visitor accepted by the element.
   **********************************************************************/
  public final void accept(Visitor visitor)
  {
    visitor.visit(this); //@ nowarn; // checking takes too long
  }
}
```

A.3 Trees

```
package patterns.visitor.tree;

import patterns.visitor.*;

/**************************************************************************
 * Base class of tree nodes.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 **************************************************************************/
public abstract class Node extends ElementBase
{
  final static int INNER = 1;
  final static int LEAF = 2;

  public final int tag;
  /*@ public invariant
    @ (tag == INNER ==> this instanceof Inner) &&
    @ (tag == LEAF  ==> this instanceof Leaf);
    @*/

  /*@ requires disjointTrees(nodes);
    @ assignable childrenM, this.tag;
    @ ensures childrenM == nodes && this.tag == tag;
    @*/
  protected Node(Node[] nodes, int tag)
  {
    super(nodes);
    this.tag = tag;
  } //@ nowarn Invariant; // checking takes too long
}

package patterns.visitor.tree;
```

```
/*****************************************************************************
 * A leaf node.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 *****************************************************************************/
public class Leaf extends Node
{
  //@ invariant numberOfChildren() == 0;

  public Leaf()
  {
    super(new Node[0], LEAF);
  }
}

package patterns.visitor.tree;

/*****************************************************************************
 * An inner node with a label and two children.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 *****************************************************************************/
public class Inner extends Node
{
  // disable checking with option -NoCheck since checker seems to loop

  private int label;

  //@ invariant numberOfChildren() == 2;

  /*@ private normal_behavior
    @ requires child1 != null && child2 != null;
    @ requires child1.isTree() && child2.isTree() && child1.disjoint(child2);
    @ assignable childrenM, this.tag, this.label;
    @ ensures childrenM != null && childrenM.length == 2;
    @ ensures childrenM[0] == child1 && childrenM[1] == child2;
    @ ensures this.tag == INNER && this.label == label;
    @*/
  public Inner(int label, Node child1, Node child2)
  {
    super(new Node[] { child1, child2 }, INNER);
    this.label = label;
  }

  /*@ private normal_behavior
    @ ensures \result == label;
    @*/
  /*@ pure @*/ public int getLabel()
  {
    return label;
  }
}
```

A.4 Tree Visitors

```
package patterns.visitor.tree;

/**********************************************************************
 * A tree visitor that prints the tree.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 **********************************************************************/
public final class NodePrinter extends NodeVisitorBase
{
  /*@ public normal_behavior
    @ assignable visitedM;
    @ ensures visitedM.length == 0;
    @*/
  public NodePrinter()
  {
    super();
  }

  public void visit(Leaf leaf)
  {
    System.out.print(".");
    childrenAccept(leaf);
  } //@ nowarn Post; // can't be established because of print()

  public void visit(Inner node)
  {
    System.out.print(node.getLabel() + "(");
    node.getChild(0).accept(this);
    System.out.print(",");
    node.getChild(1).accept(this);
    System.out.print(")");
  } //@ nowarn Post; // checking takes too long
}

package patterns.visitor.tree;

/**********************************************************************
 * A tree visitor that computes the sum of the labels of all inner nodes.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 **********************************************************************/
public final class NodeAdder extends NodeVisitorBase
{
  private int sum; //@ in visitedM;

  /*@ public normal_behavior
    @ assignable visitedM;
    @ ensures visitedM.length == 0 && getSum() == 0;
    @*/
  public NodeAdder()
  {
    super();
```

```
    sum = 0;
  }

  /*@ also public normal_behavior
    @ assignable visitedM;
    @ ensures getSum() == 0;
    @*/
  public void reset()
  {
    super.reset();
    sum = 0;
  }

  /*@ public normal_behavior
    @ ensures \result == getSum();
    @ also private normal_behavior
    @ ensures \result == sum;
    @*/
  /*@ pure @*/ public int getSum()
  {
    return sum;
  }

  /*@ also public normal_behavior
    @ requires node != null && node.isTree();
    @ requires hasSum(visitedM.length-1);
    @ assignable \everything;
    @ ensures hasSum(visitedM.length);
    @*/
  public void visit(Inner node)
  {
    sum = sum + node.getLabel();
    childrenAccept(node); //@ nowarn; // checking takes too long otherwise
  } //@ nowarn Post; // postcondition can't be established

  /*@ public normal_behavior
    @ requires 0 <= pos && pos < visitedM.length;
    @ ensures \result == true <==>
    @   getSum() ==
    @     (\sum int i; 0 <= i && i < pos;
    @        (visitedM[i] instanceof Inner) ?
    @          ((Inner)visitedM[i]).getLabel() : 0);
    @ public pure model boolean hasSum(int pos);
    @*/
}

package patterns.visitor.tree;

/**********************************************************************
 * Testing two tree visitors.
 * @author Wolfgang Schreiner &lt;Wolfgang.Schreiner@risc.jku.at&gt;
 * @see <a href="http://en.wikipedia.org/wiki/Visitor_pattern">visitor pattern</a>
 **********************************************************************/
public class Main
```

```
{
  public static void main(String[] args)
  {
    Inner tree = new Inner(1, new Inner(2, new Leaf(),
        new Inner(3, new Leaf(), new Leaf())),
        new Inner(4, new Leaf(), new Leaf()));

    // printing
    NodePrinter printer = new NodePrinter();
    tree.accept(printer);
    System.out.println("\n" + printer.check(tree));

    // adding
    NodeAdder adder = new NodeAdder();
    tree.accept(adder);
    System.out.println(adder.getSum());
    System.out.println(adder.check(tree));
  }
}
```