Visitor Pattern

CS356 Object-Oriented Design and Programming
http://cs356.yusun.io
November 5, 2014

Yu Sun, Ph.D.
http://yusun.io
yusun@csupomona.edu

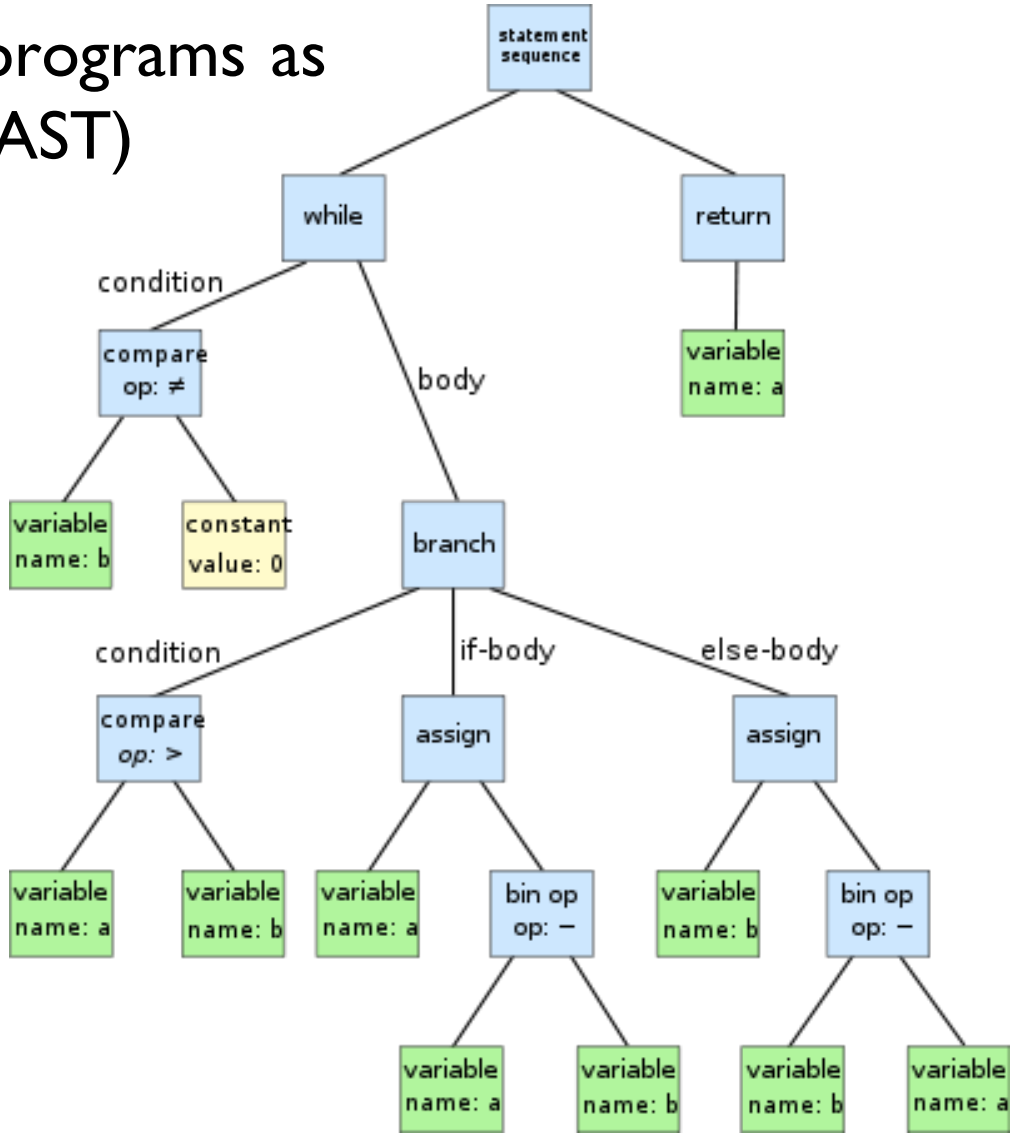
CAL POLY POMONA

Visitor

- *Intent*
  - Represent an operation on elements of an object structure
  - Lets you define a new operation without changing classes of the elements on which it operates

Motivating Example

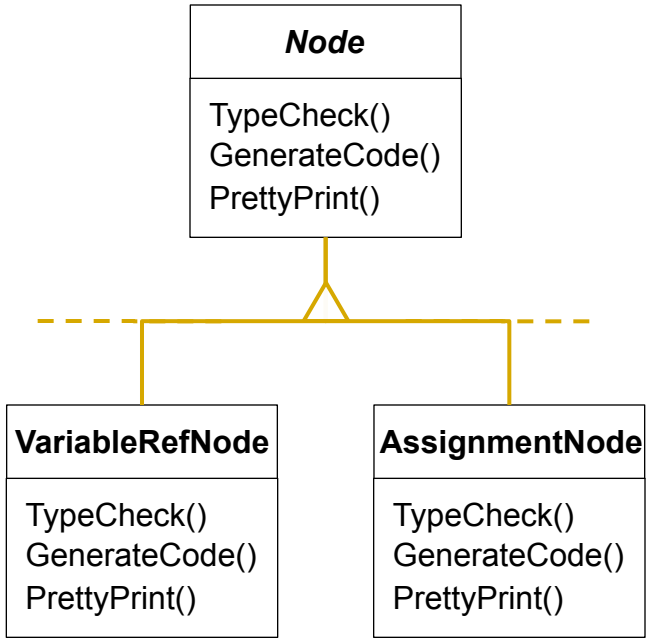
- Compiler represents programs as abstract syntax trees (AST)




```
While b!=0
  if a > b
    a := a - b
  else
    b := b - a
return a
```

Motivating Example

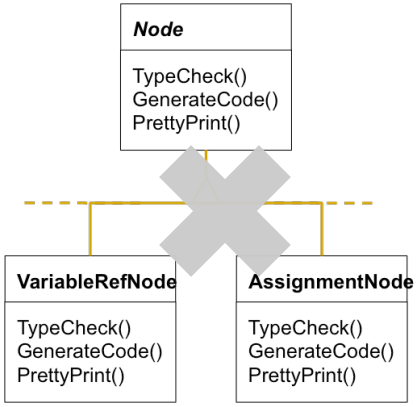
- Compiler represents programs as abstract syntax trees (AST)
- Need to perform operations on AST for semantic analysis, code-generation, pretty-printing, etc.
- AST has different types of nodes for variable reference, assignment, operator, etc.
- Code for an operation is specific to the node type

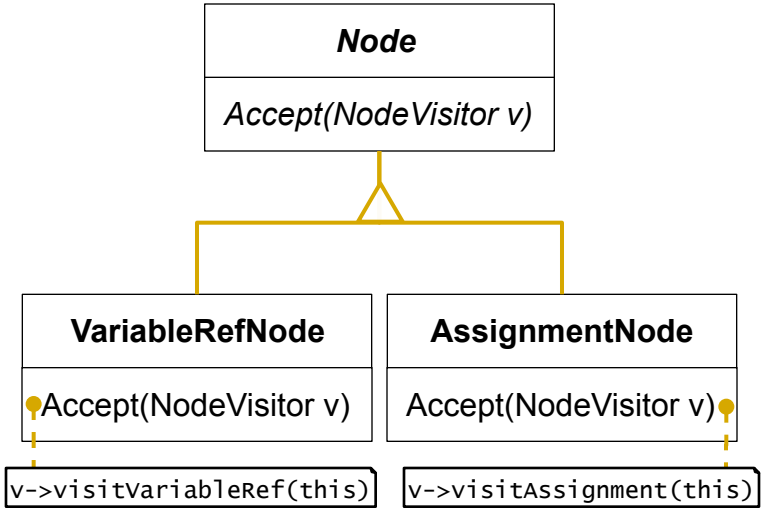


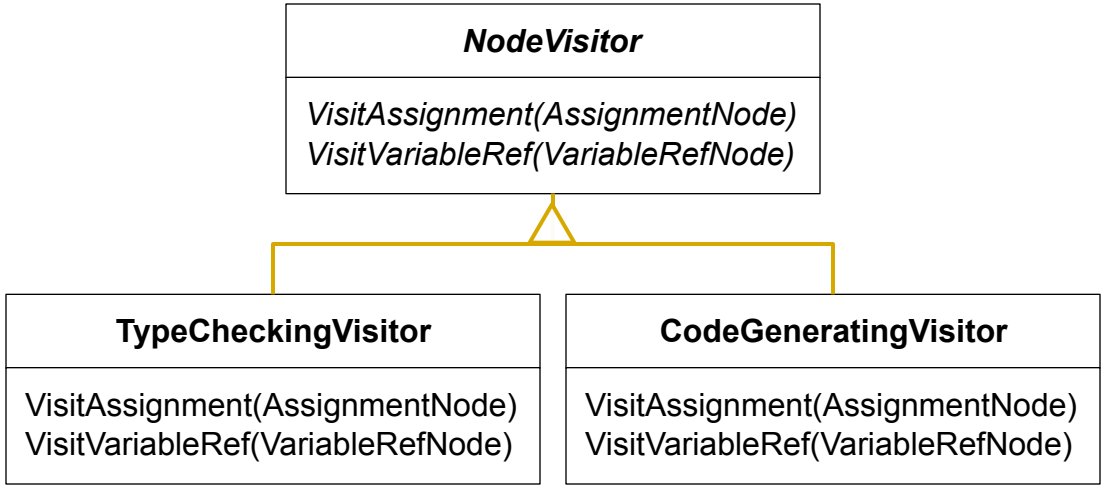
Motivation

- Adding operation code to each node type has the following drawbacks
  - Code for different unrelated operations are mixed together as methods on a single node
  - Adding a new operation would mean changing and recompiling all the node classes
- Better Approach
  - Node classes independent of the operations applied to them

Visitor Pattern Solution

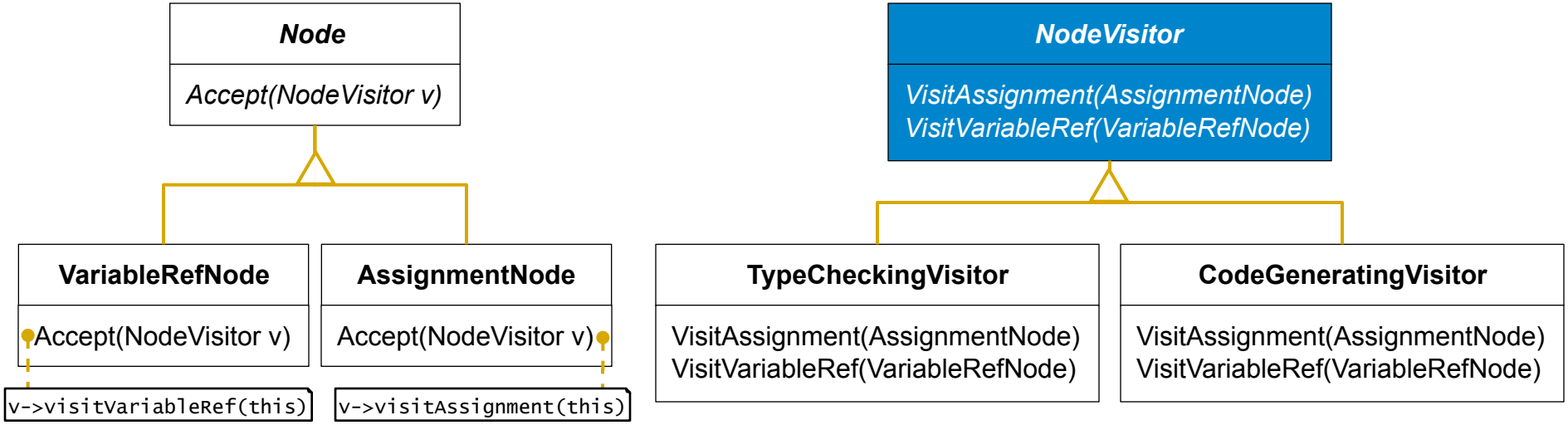






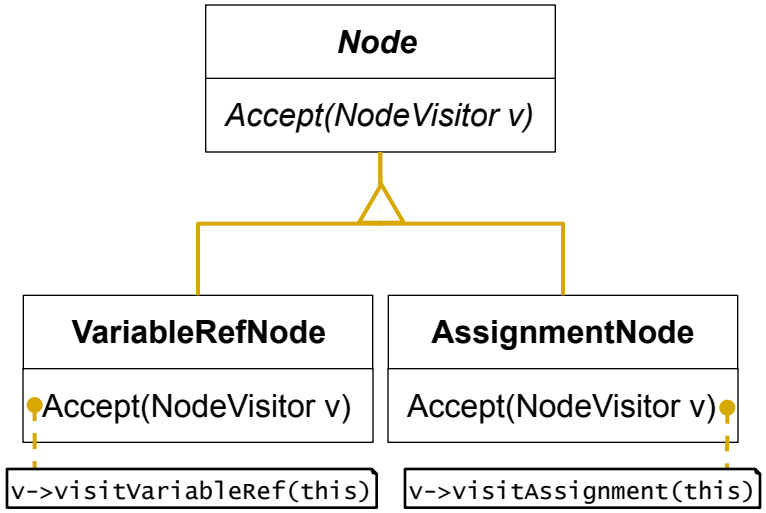
Visitor (NodeVisitor)

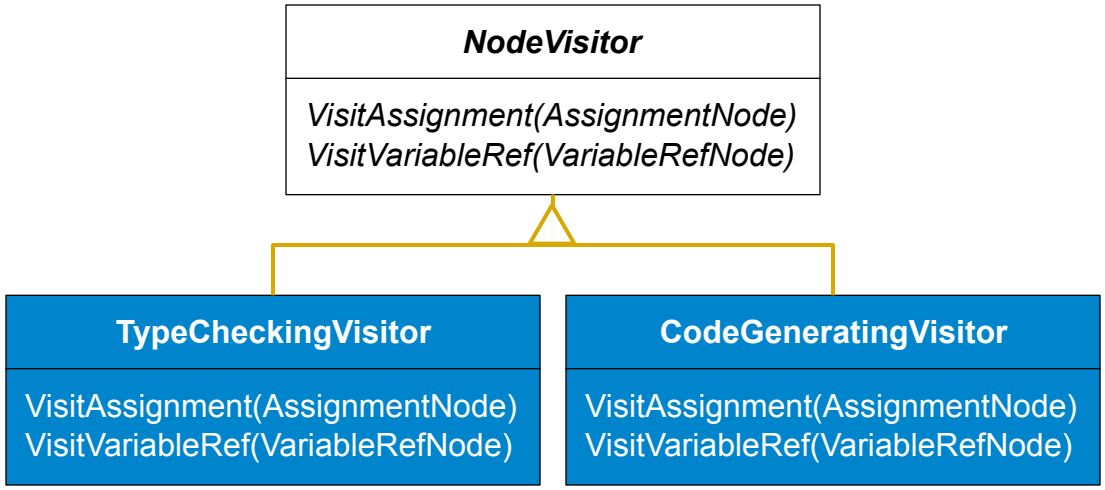
- Visitor (NodeVisitor) is an abstract class that has a different method for each type of object on which it operates



TypeCheckingVisitor / CodeGeneratingVisitor

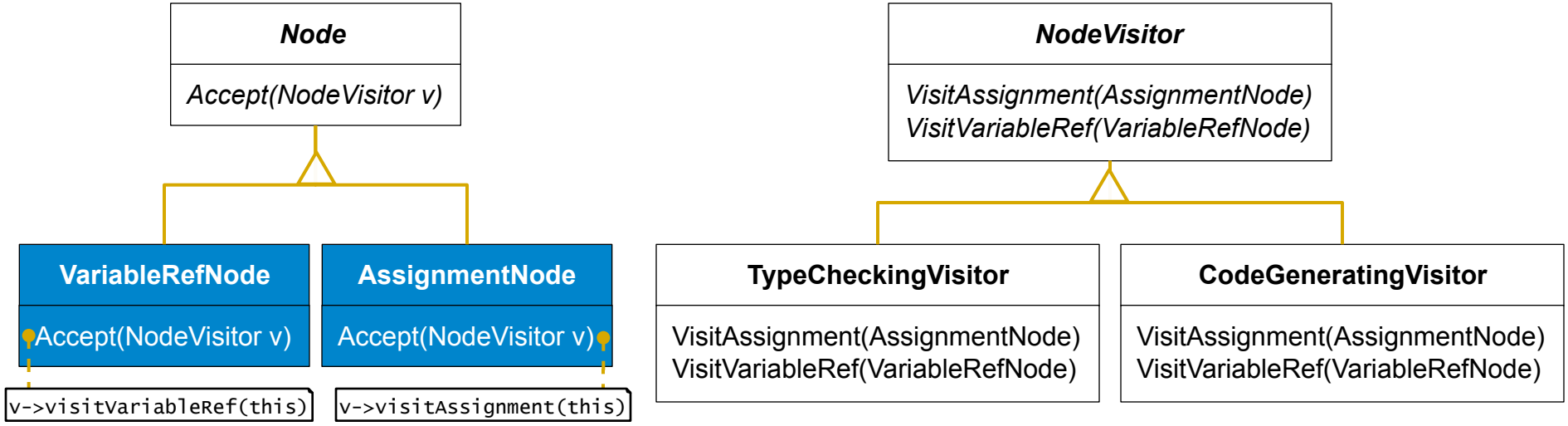
- Each operation is a subclass of NodeVisitor and overloads the type-specific methods





VariableRefNode / AssignmentNode

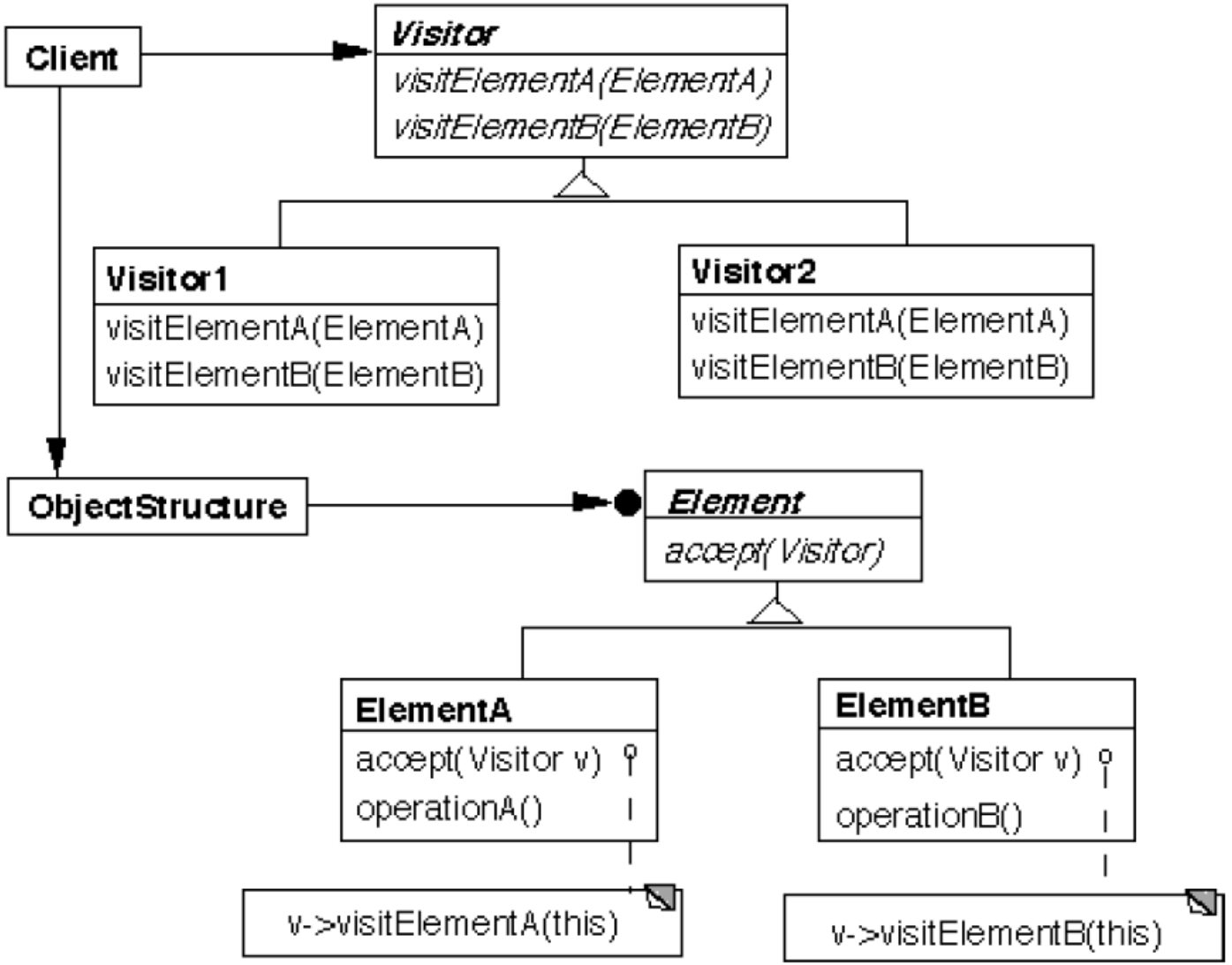
- Objects that are operated on accept a NodeVisitor
  - Call back their type-specific method
  - Pass themselves as operands
- Objects are independent of the operations and new operations can be added without modifying the object types



Applicability

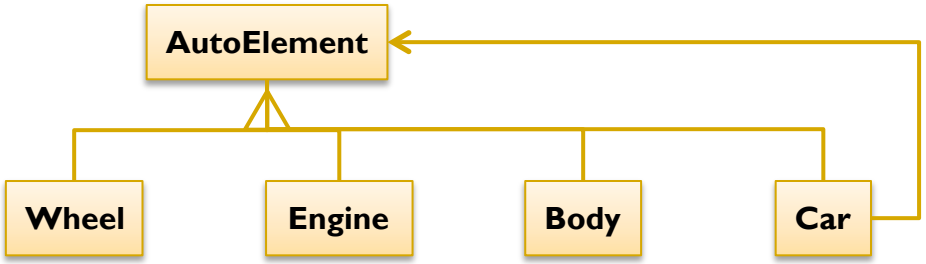
- When operations of an object structure depend on their concrete classes
- Many unrelated operations need to be performed
  - Want to keep related operations together
- The classes defining the object structure rarely change (structure is stable)
  - Want to define new operations over the structure
- Caveat: If the classes of object structure change often, it is better to define the operations in those classes

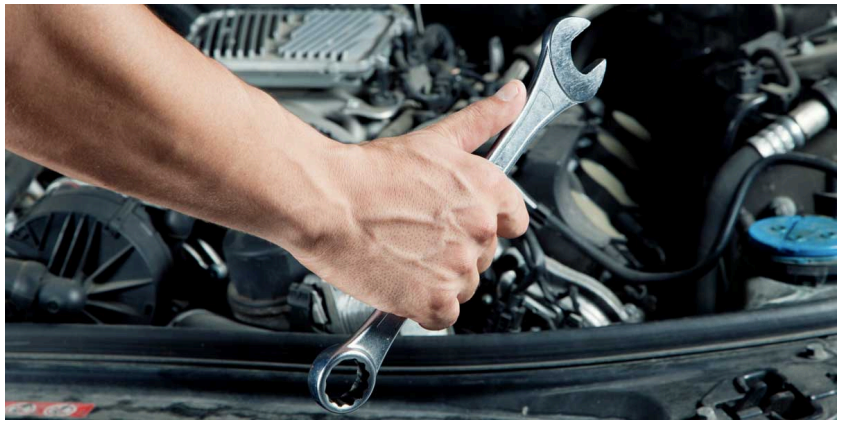
Structure


Client
Visitor
visitElementA(ElementA)
visitElementB(ElementB)
Visitor1
visitElementA(ElementA)
visitElementB(ElementB)
Visitor2
visitElementA(ElementA)
visitElementB(ElementB)
ObjectStructure
Element
accept(Visitor)
ElementA
accept(Visitor v)
operationA()
ElementB
accept(Visitor v)
operationB()
v->visitElementA(this)
v->visitElementB(this)

Example - Car

- Do maintenance
- Check recall
- Upgrade
- …





Example – Equipment and Card

```
abstract class Equipment {
   abstract double netPrice();
   abstract double discountPrice();

   abstract void accept(Visitor vis);
}
```

```
class Card extends Equipment {
  double netPrice() {
    return 10.00;
  }

  double discountPrice() {
    return 2.00;
  }

  void accept(Visitor vis) {
    vis.atCard(this);
  }
}
```

CompositeEquip and Cabinet

```
abstract class CompositeEquip extends Equipment {
  Equipment[] parts;

  CompositeEquipment() { ... }

  double netPrice() { ...  }
  double discountPrice() { ... }

  void accept(Visitor vis) {
    for(int i=0; i<parts.length; i++)
      parts[i].accept(vis);
  }
}
```

```
class Cabinet extends CompositeEquip {
  Cabinet() { ... }
  void accept(Visitor vis) {
    vis.atCabinet(this);
    super.accept(vis);
  }
}
```

Visitor, PricingVisitor, and InventoryVisitor

```
interface class Visitor {
  void atCard(Card e);
  void atDrive(Drive e);
  void atBoard(Board e);
  void atCabinet(Cabinet e);
  void atBus(Bus e);
  void atChassis(Chassis e);
}
```

```
class PricingVisitor implements Visitor {
  private float total;
  PricingVisitor() { total = 0; }
  float value() { return total; }
  void atCard(Card e) { total += e.netPrice(); }
  void atDrive(Drive e) { total += e.netPrice(); }
  void atBoard(Board e) { total += e.netPrice(); }
  void atCabinet(Cabinet e) { total += e.discountPrice(); }
  void atBus(Bus e) { total += e.discountPrice(); }
  void atChassis(Chassis e) { total += e.discountPrice(); }
}
```

```
class InventoryVisitor implements Visitor {
  private Vector inv;
  InventoryVisitor() { inv = new Vector(10,5); }
  int size() { return inv.size(); }
  void atCard(Card e) { inv.addElement(e); }
  void atDrive(Drive e) { inv.addElement(e); }
  void atBoard(Board e) { inv.addElement(e); }
  void atCabinet(Cabinet e) { inv.addElement(e); }
  void atBus(Bus e) { inv.addElement(e); }
  void atChassis(Chassis e) { inv.addElement(e); }
}
```

TestVisitor

```
public class TestVisitor {

  public static void main(String argv[]) {
    Equipment equip = new Cabinet();
    // add items to Cabinet

    PricingVisitor pri = new PricingVisitor();
    equip.accept(pri);
    System.out.println(“Pricing” + pri.value());

    InventoryVisitor inv = new InventoryVisitor();
    equip.accept(inv);
    System.out.println("Number elements" + inv.size());
  }
}
```

Consequences

- + Addition of new operations is easy
  - New operations can be created by adding a new Visitor
- + Gathers related operations together
  - All operation-related code is in the Visitor
  - Code for different operations in separate Visitor subclasses
  - Unrelated operations not mixed together in object classes

- × Adding a new concrete type in object structure is hard
  - Each Visitor has to be recompiled with an appropriate method for the new type