Lecture 15 - July 3

Design Patterns: Composite, Visitor

Tracing Composite
Visitor: Design Rationales
Visitor: Tracing Double Dispatch

Composite Pattern: Implementation

```
public interface Equipment {
  public String name();
  public double price(); /* uniform access */
}
```

uniform access

```
public abstract class Composite<E> {
  protected List<E> children;

  public void add(E child) {
    children.add(child); /* polymorphism */
  }
}
```

```
public abstract class BaseEquipment implements Equipment {
  private String name;
  private double price;
  public BaseEquipment(String name, double price) {
    this.name = name; this.price = price;
  }
  public String name() { return this.name; }
  public double price() { return this.price; }
}
```

access !

```
public abstract class CompositeEquipment
  extends Composite<Equipment>
  implements Equipment
{
  private String name;
  public CompositeEquipment(String name) {
    this.name = name;
    this.children = new ArrayList<>();
  }
  public String name() { return this.name; }
  public double price() {
    double result = 0.0;
    for(Equipment child : this.children) {
      result = result + child.price(); /* dynamic binding */
    }
    return result;
  }
}
```

DT can be either Base or Composite

uniform access

```
public class VideoCard extends BaseEquipment {
  public VideoCard(String name, double price) {
    super(name, price);
  }
}
```

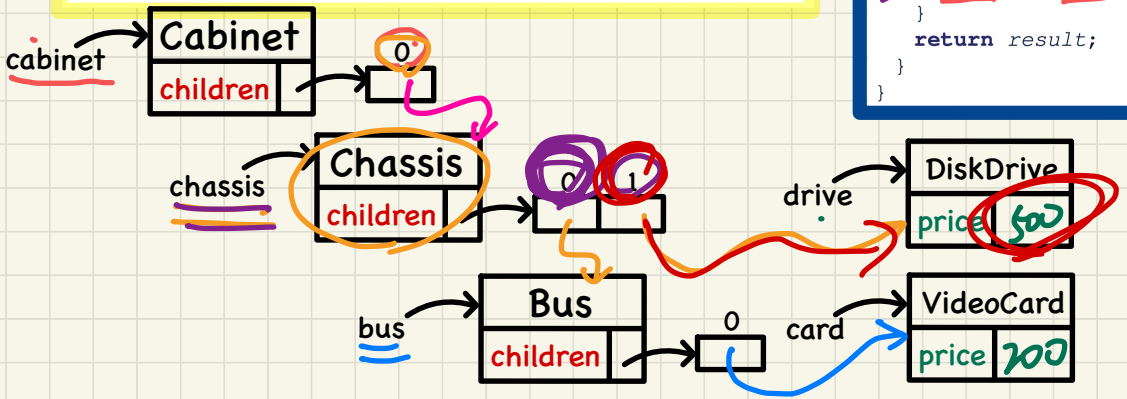
```
public class Chassis extends CompositeEquipment {
  public Chassis(String name) {
    super(name);
  }
}
```

Composite Pattern: Testing

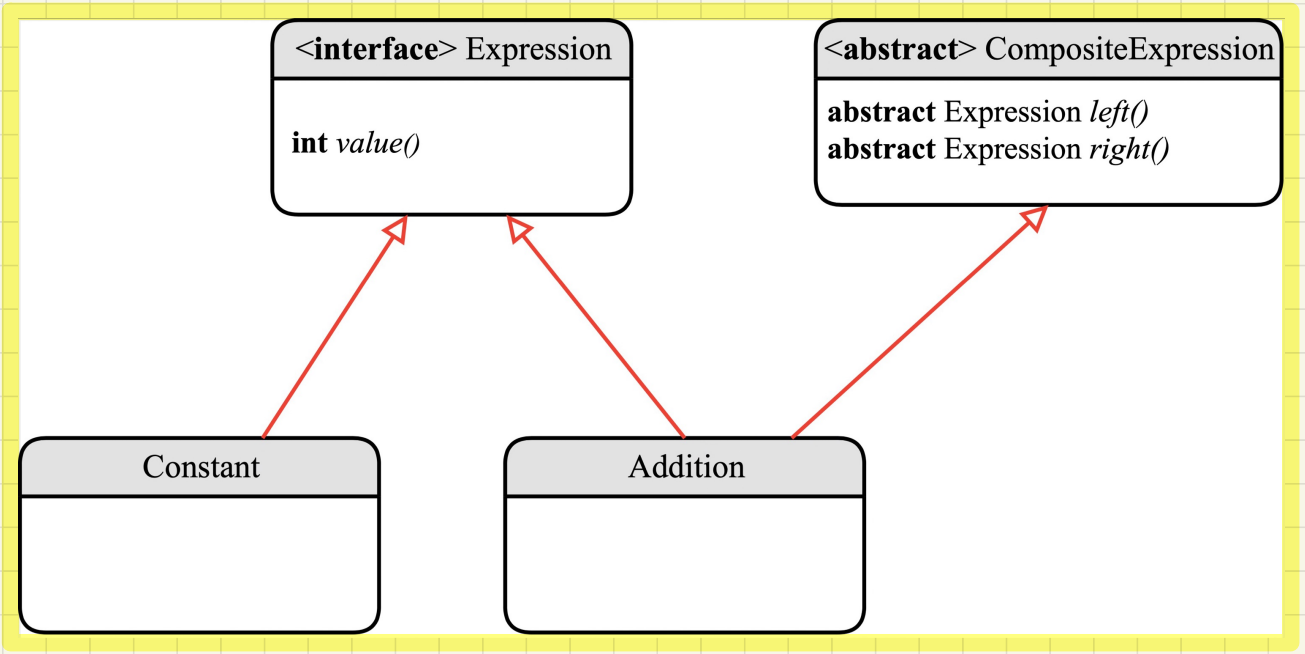
```java
@Test
public void test_equipment() {
  Equipment card, drive;
  Bus bus;
  Cabinet cabinet;
  Chassis chassis;

  card = new VideoCard("16Mbs Token Ring", 200);
  drive = new DiskDrive("500 GB harddrive", 500);
  bus = new Bus("MCA Bus");
  chassis = new Chassis("PC Chassis");
  cabinet = new Cabinet("PC Cabinet");
  bus.add(card);
  chassis.add(bus);
  chassis.add(drive);
  cabinet.add(chassis);

  assertEquals(700.00, cabinet.price(), 0.1);
}
```

```java
public abstract class BaseEquipment implements Equipment {
  private String name;
  private double price;
  public BaseEquipment(String name, double price) {
    this.name = name; this.price = price;
  }
  public String name() { return this.name; }
  public double price() { return this.price; }
}
```

```java
public abstract class CompositeEquipment
  extends Composite<Equipment>
  implements Equipment
{
  private String name;
  public CompositeEquipment(String name) {
    this.name = name;
    this.children = new ArrayList<>();
  }
  public String name() { return this.name; }
  public double price() {
    double result = 0.0;
    for(Equipment child : this.children) {
      result = result + child.price(); /* dynamic binding */
    }
    return result;
  }
}
```



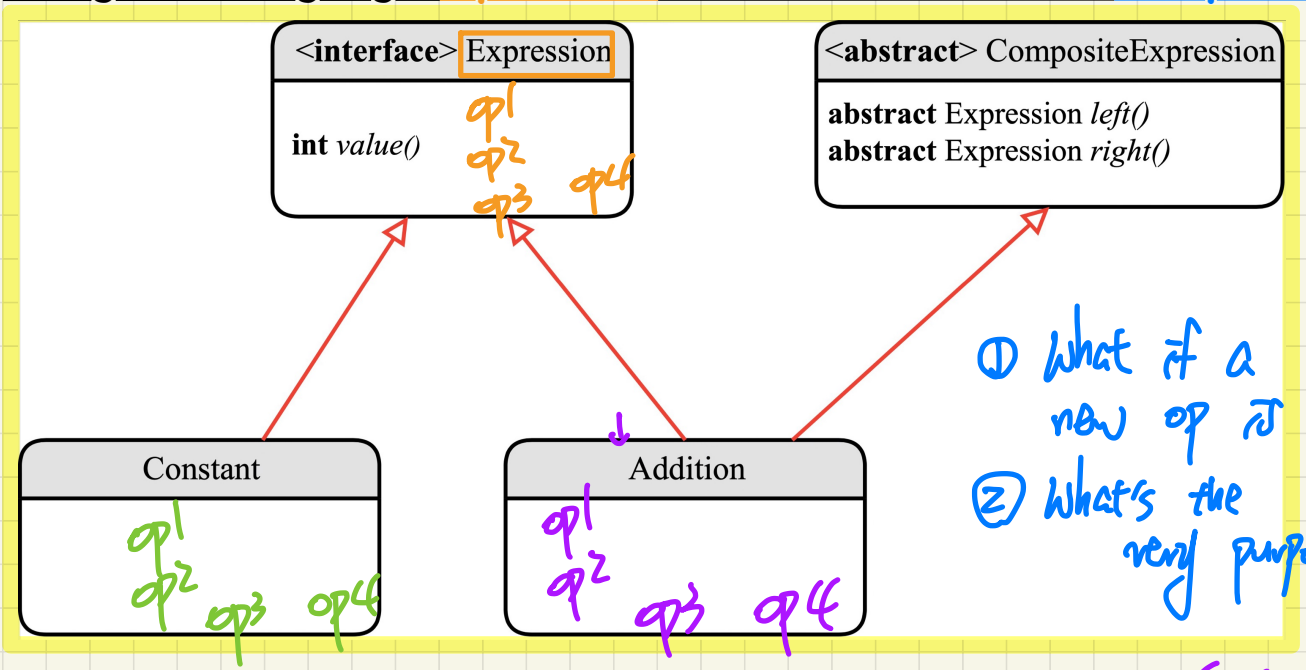
Design of Language Structure: Composite Pattern



Q: How to construct a composite object representing "341 + 2"?

Q: How to extend the design to include variables and subtractions?

Design of Language Operation: How to Extend the Composite Pattern?



Structure

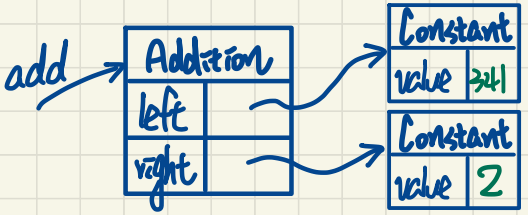
① What if a new op is needed?

② What's the very purpose of a class? (superman class)

op1 evaluate
op2 print_prefix
op3 print_postfix
op4 type_check

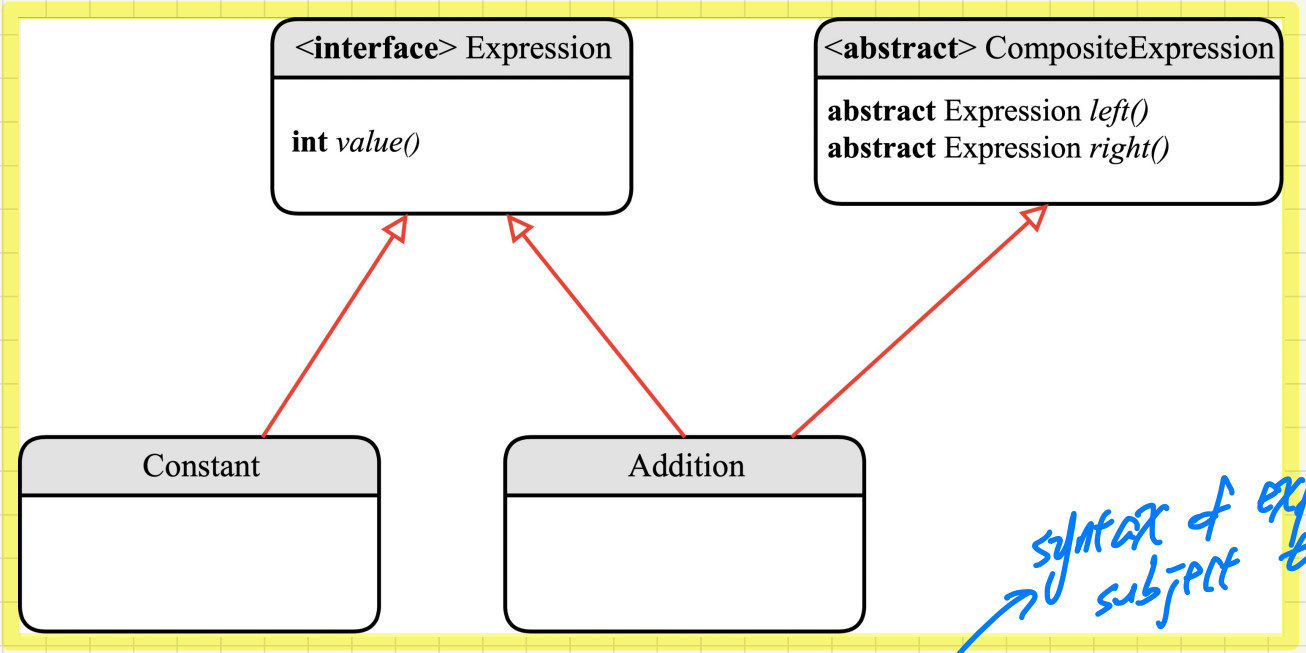
Operations

op4.



modularity

Design of a Language Application: Open-Closed Principle



Structure

syntax of exp. subject to changes

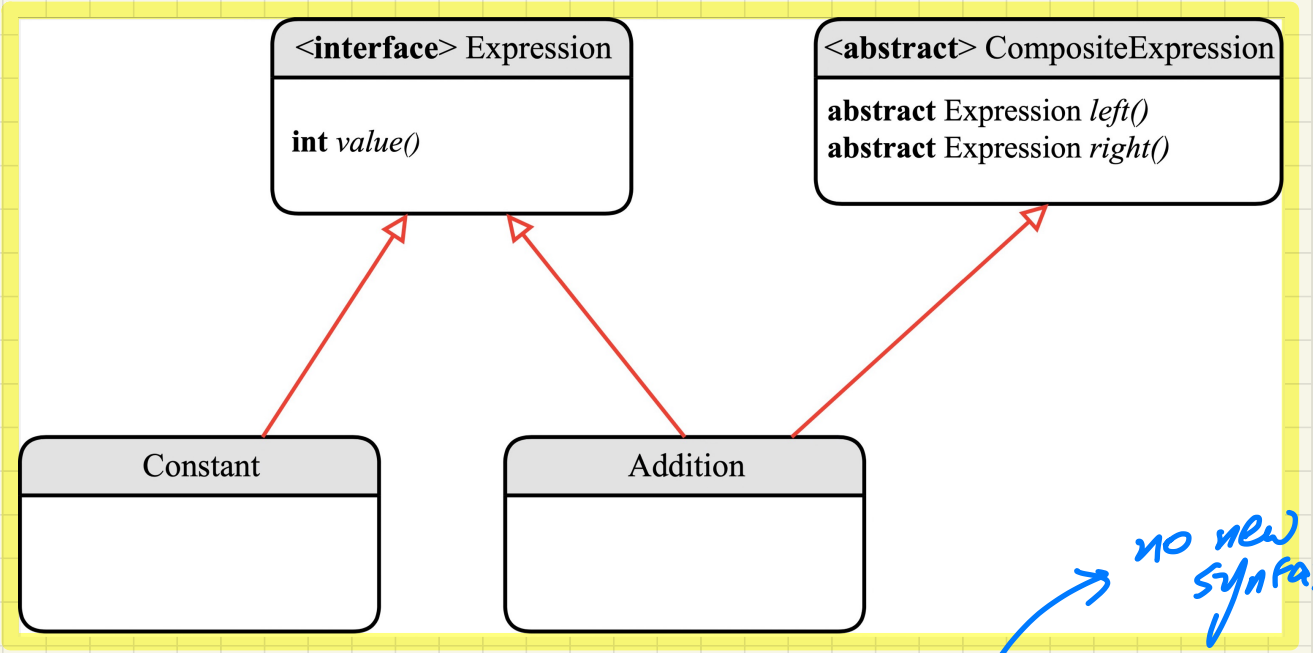
evaluate
print_prefix
print_postfix
type_check

Operations

	Structure	Operations
Alternative 1	Open	Closed
Alternative 2	Closed	Open

→ list of supported ops. is fixed

Design of a Language Application: Open-Closed Principle



Structure

evaluate
print_prefix
print_postfix
type_check

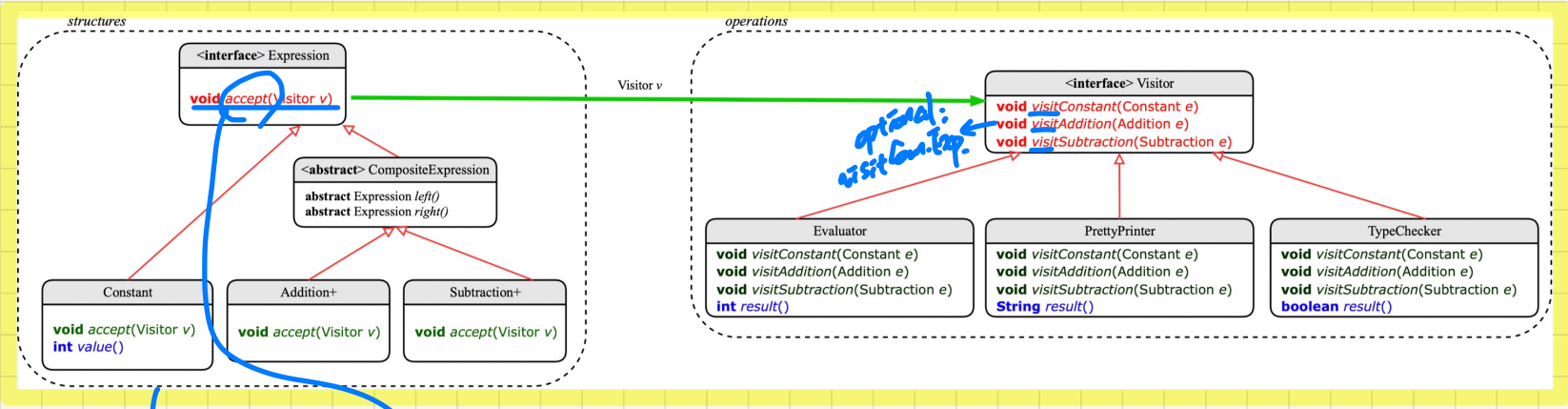
Operations

	Structure	Operations
Alternative 1	Open	Closed
Alternative 2	Closed	Open

no new syntax

keep adding new ops.

Visitor Design Pattern: Architecture

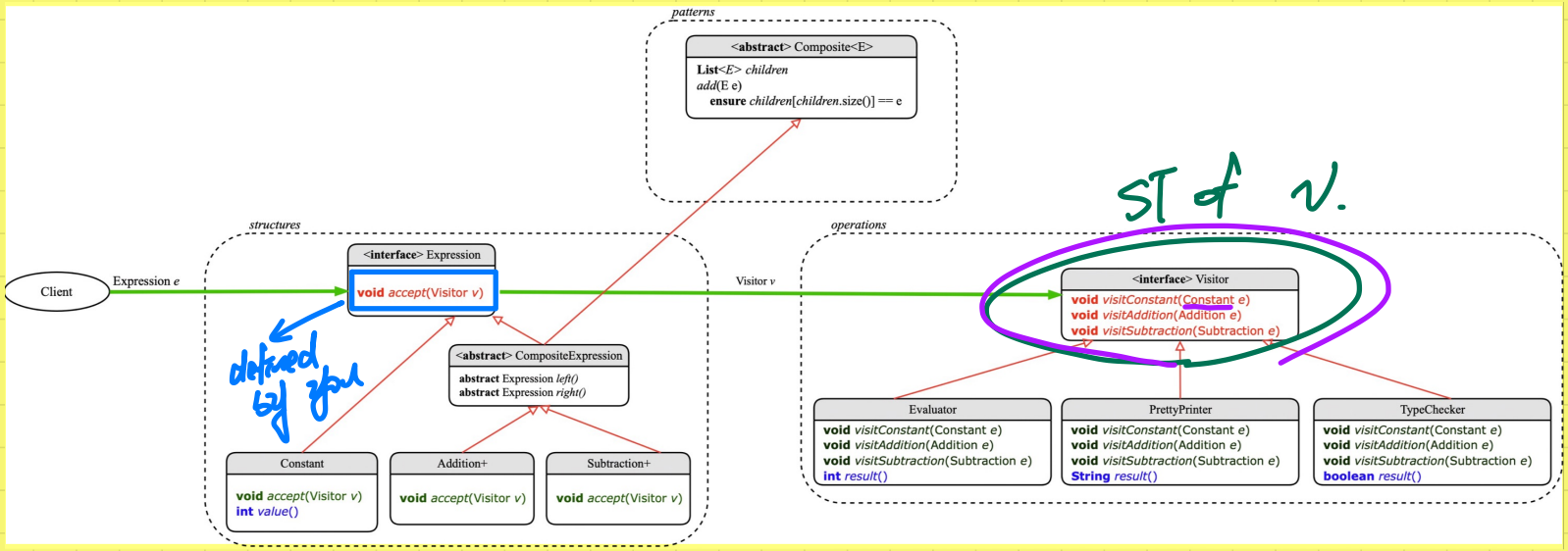


Composite.

Expression e = DT.

e.accept(visitor)

Visitor Design Pattern: Architecture




```
1  @Test
2  public void test_expression_evaluation() {
3    CompositeExpression add;
4    Expression c1, c2;
5    Visitor v;
6    c1 = new Constant(1); c2 = new Constant(2);
7    add = new Addition(c1, c2);
8    v = new Evaluator();
9    add.accept(v);
10   assertEquals(3, ((Evaluator) v).result());
11 }
```

static type

dynamic type 1+2.

root of AST to start processing

How to Use Visitors

Can I write v.result()?

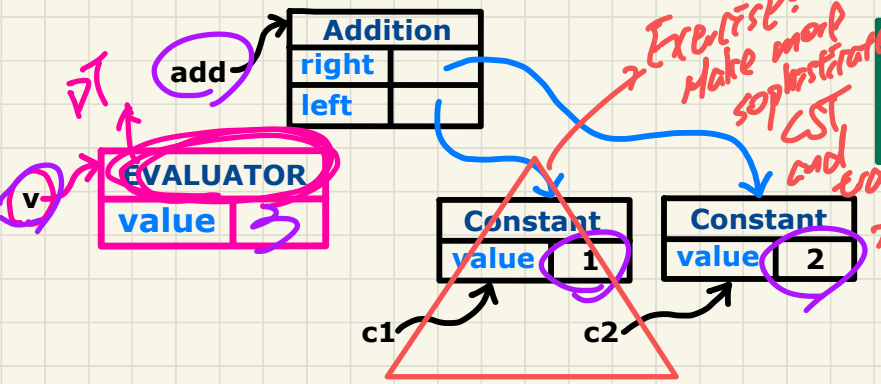
↳ ST of v is Visitor, which does not support result()

Visitor Design Pattern: Implementation

```
1  @Test
2  public void test_expression_evaluation() {
3    CompositeExpression add;
4    Expression c1, c2;
5    Visitor v;
6    c1 = new Constant(1); c2 = new Constant(2);
7    add = new Addition(c1, c2);
8    v = new Evaluator();
9    add.accept(v);
10   assertEquals(3, ((Evaluator) v).result());
11 }
```

Visualizing Line 3 to Line 7

Executing Composite and Visitor Patterns at Runtime



Tracing add.accept(v)

Double Dispatch

1st dispatch: DT of add is Addition → version of accept in Addition is evoked.

```
public interface Visitor {
  public void visitConstant(Constant e);
  public void visitAddition(Addition e);
  public void visitSubtraction(Subtraction e);
}
```

```
public class Constant implements Expression {
  ...
  public void accept(Visitor v) {
    v.visitConstant(this);
  }
}
```

→ Evaluator version

```
public class Addition extends CompositeExpression {
  ...
  public void accept(Visitor v) {
    v.visitAddition(this);
  }
}
```

this: Add

2nd dispatch: DT of v is Evaluator → version of visitAddition in Evaluator is invoked

```
public class Evaluator implements Visitor {
  private int result;
  ...
  public void visitConstant(Constant e) {
    this.result = e.value();
  }
  public void visitAddition(Addition e) {
    Evaluator evalL = new Evaluator();
    Evaluator evalR = new Evaluator();
    e.getLeft().accept(evalL);
    e.getRight().accept(evalR);
    this.result = evalL.result() + evalR.result();
  }
}
```

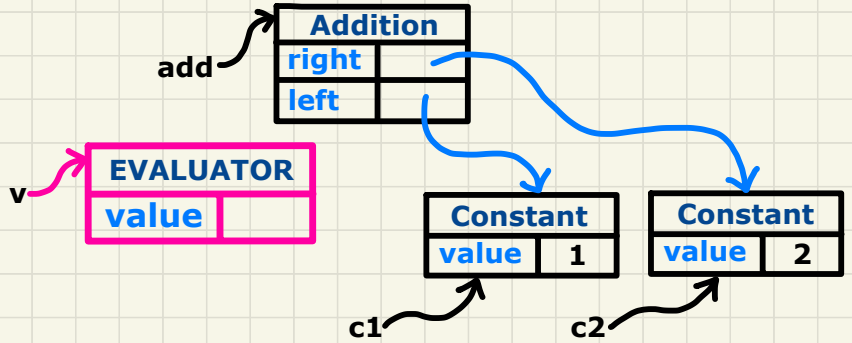
DT: constant

add e → + (L, R)

double-dispatch double-dispatch

3 1 2

Executing Composite and Visitor Patterns at Runtime



Tracing add.accept(v)
Double Dispatch

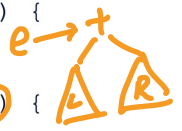
DT DT

```
public class Constant implements Expression {
  ...
  public void accept(Visitor v) {
    v.visitConstant(this);
  }
}
```

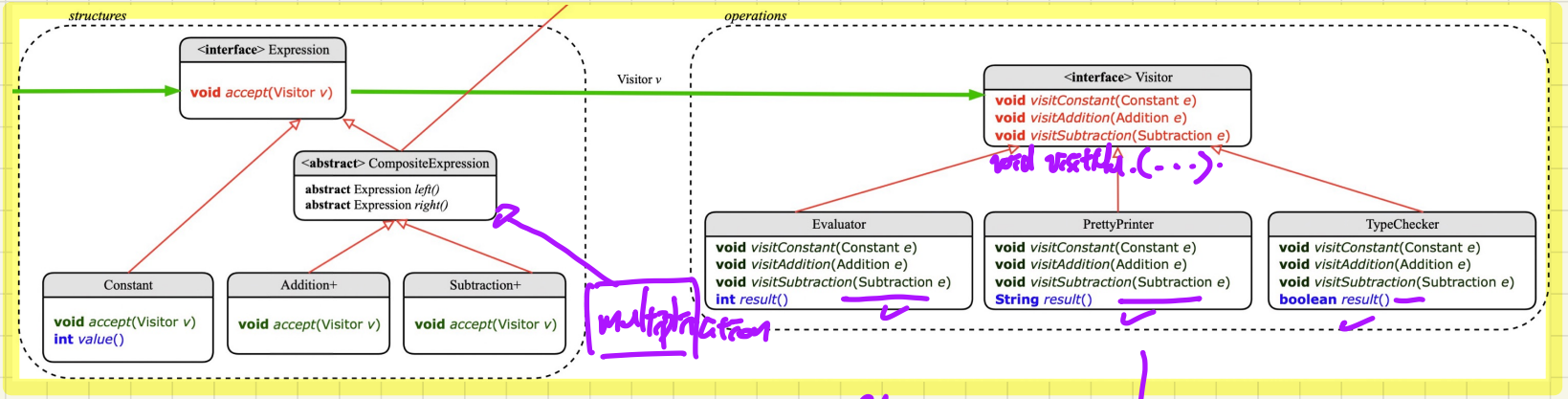
```
public class Addition extends CompositeExpression {
  ...
  public void accept(Visitor v) {
    v.visitAddition(this);
  }
}
```

```
public interface Visitor {
  public void visitConstant(Constant e);
  public void visitAddition(Addition e);
  public void visitSubtraction(Subtraction e);
}
```

```
public class Evaluator implements Visitor {
  private int result;
  ...
  public void visitConstant(Constant e) {
    this.result = e.value();
  }
  public void visitAddition(Addition e) {
    Evaluator evalL = new Evaluator();
    Evaluator evalR = new Evaluator();
    e.getLeft().accept(evalL);
    e.getRight().accept(evalR);
    this.result = evalL.result() + evalR.result();
  }
}
```



Visitor Pattern: Open-Closed and Single-Choice Principles



structure: open
operation: closed

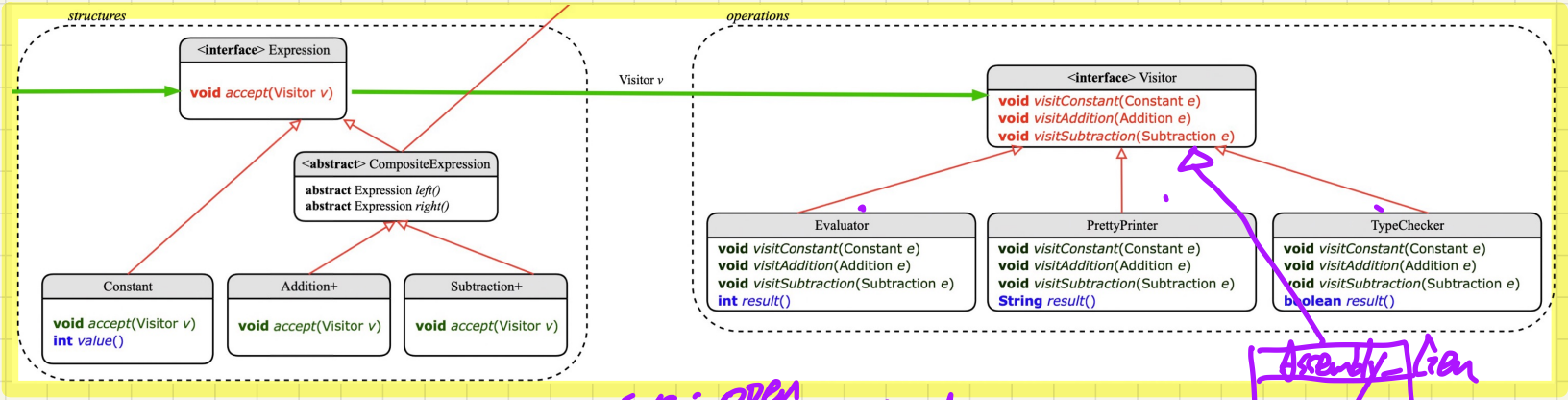
violates SCP.

What if a **new language construct** is added?

↳ unsuitable for visitor pattern

If the **visitor pattern** is adopted, what should be **closed**?

Visitor Pattern: Open-Closed and Single-Choice Principles



operations: open
structures: closed

What if a new language operation is added?

If the visitor pattern is adopted, what should be open?

To this single place, implement all visit methods
↳ SCP satisfied