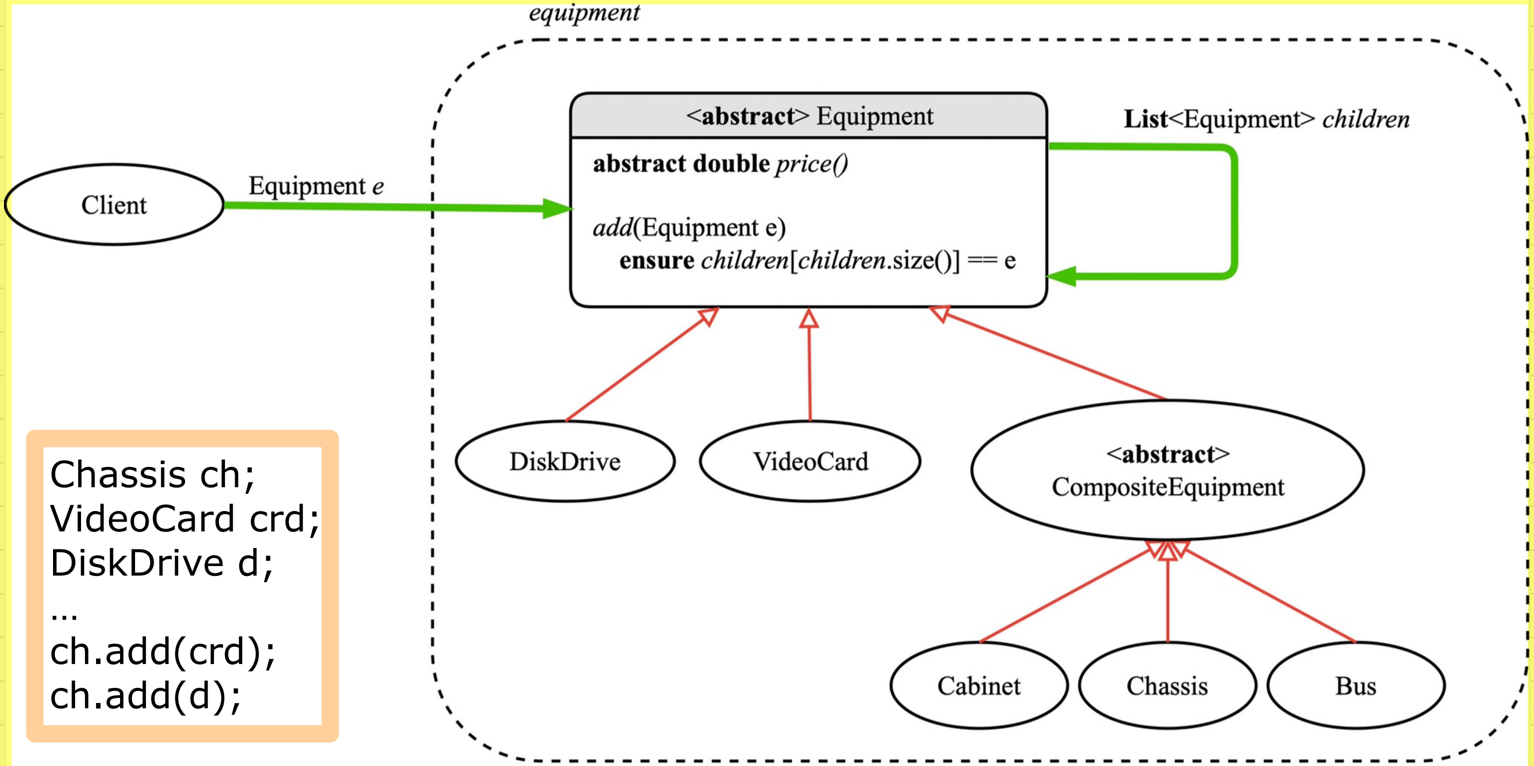
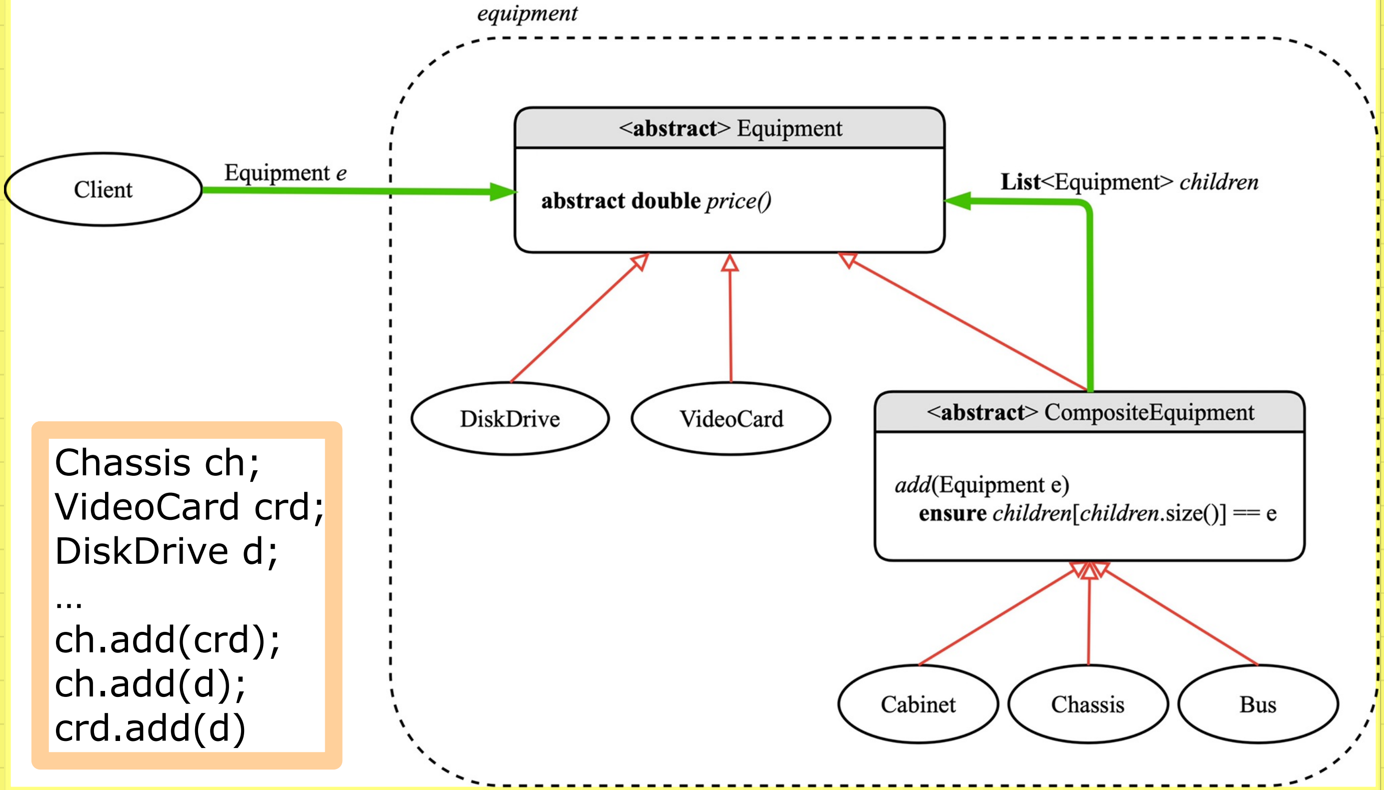
First Design Attempt



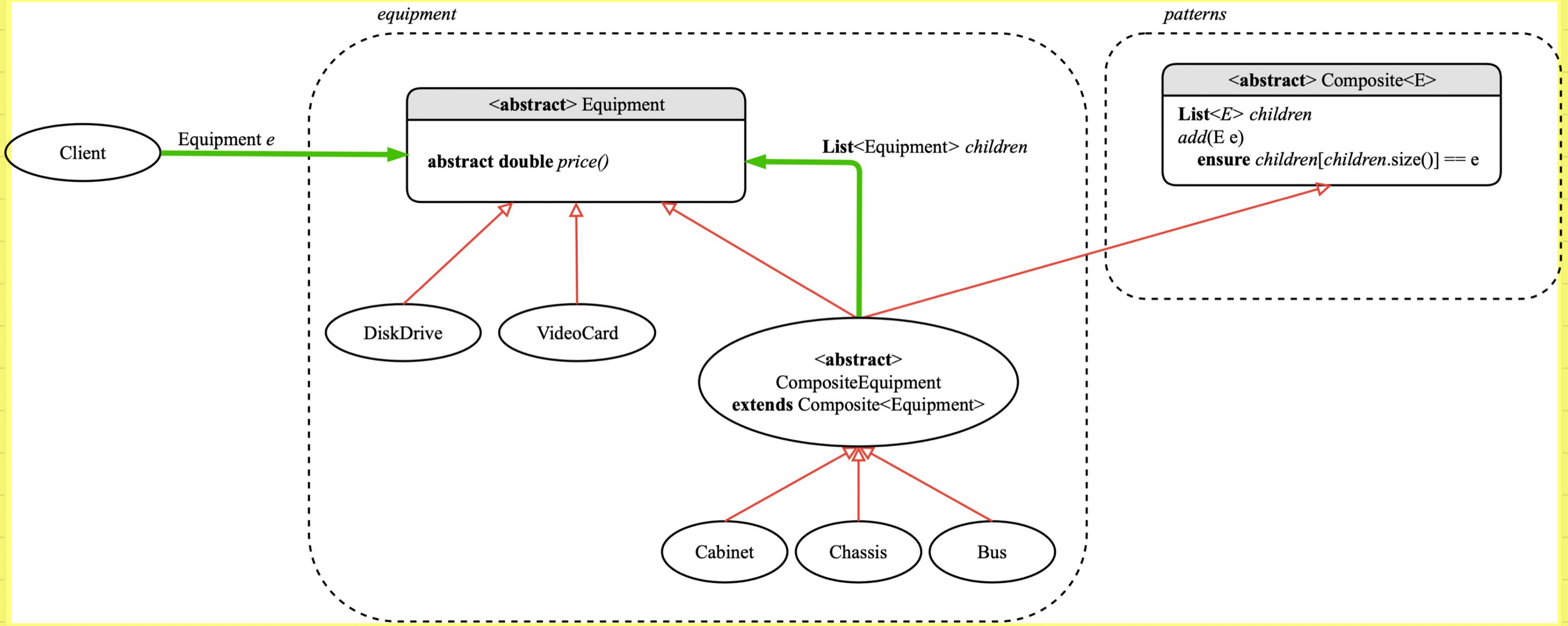
```
Chassis ch;
VideoCard crd;
DiskDrive d;
…
ch.add(crd);
ch.add(d);
```

Second Design Attempt

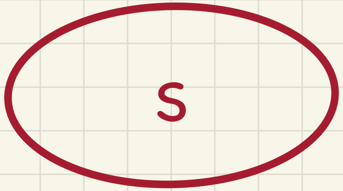

equipment
Client
Equipment *e*
<**abstract**> Equipment
abstract double *price()*
List<Equipment> *children*
DiskDrive
VideoCard
<**abstract**> CompositeEquipment
add(Equipment e)
ensure *children*[*children*.size()] == e
Cabinet
Chassis
Bus

```
Chassis ch;
VideoCard crd;
DiskDrive d;
…
ch.add(crd);
ch.add(d);
crd.add(d)
```

Third Design Attempt

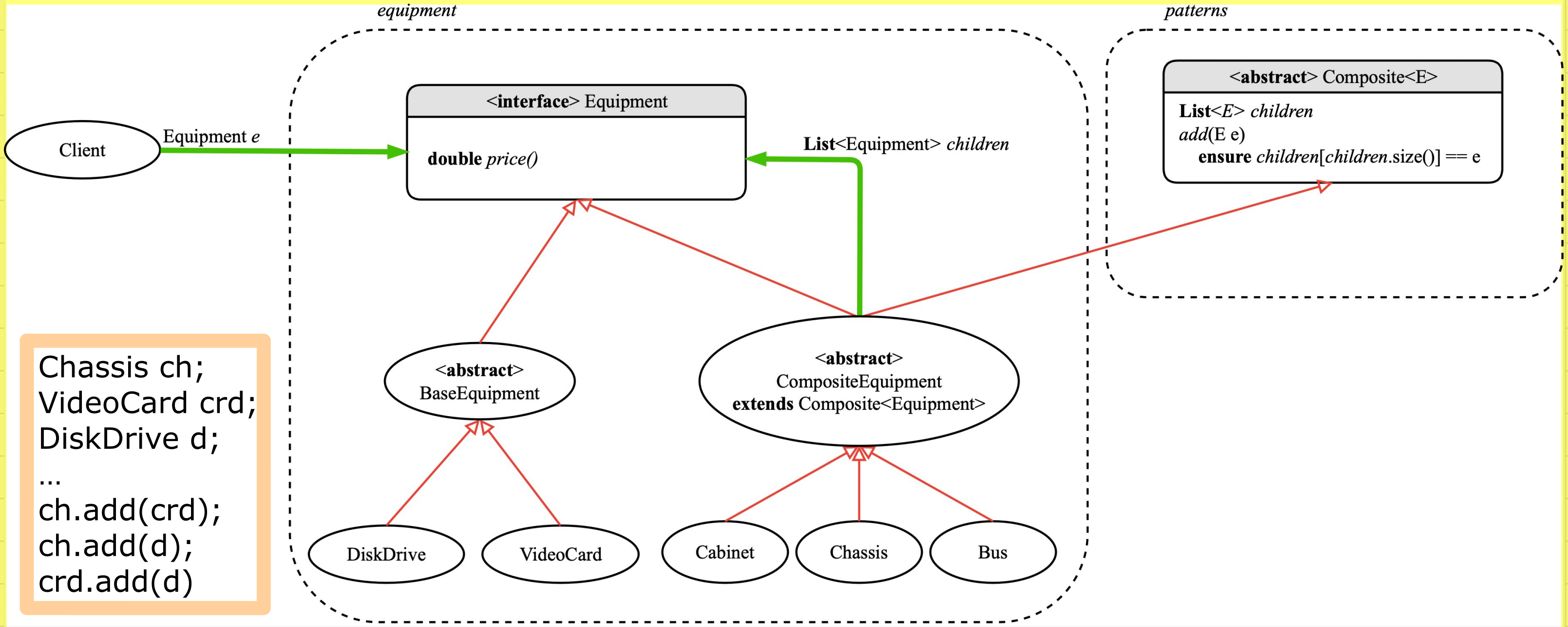

equipment
patterns
Client
Equipment *e*
<**abstract**> Equipment
abstract double *price()*
List<Equipment> *children*
<**abstract**> Composite<E>
List<*E*> *children*
add(E e)
ensure *children*[*children*.size()] == e
DiskDrive
VideoCard
<**abstract**>
CompositeEquipment
extends Composite<Equipment>
Cabinet
Chassis
Bus

Multiple Inheritance in Java: Diamond Problem



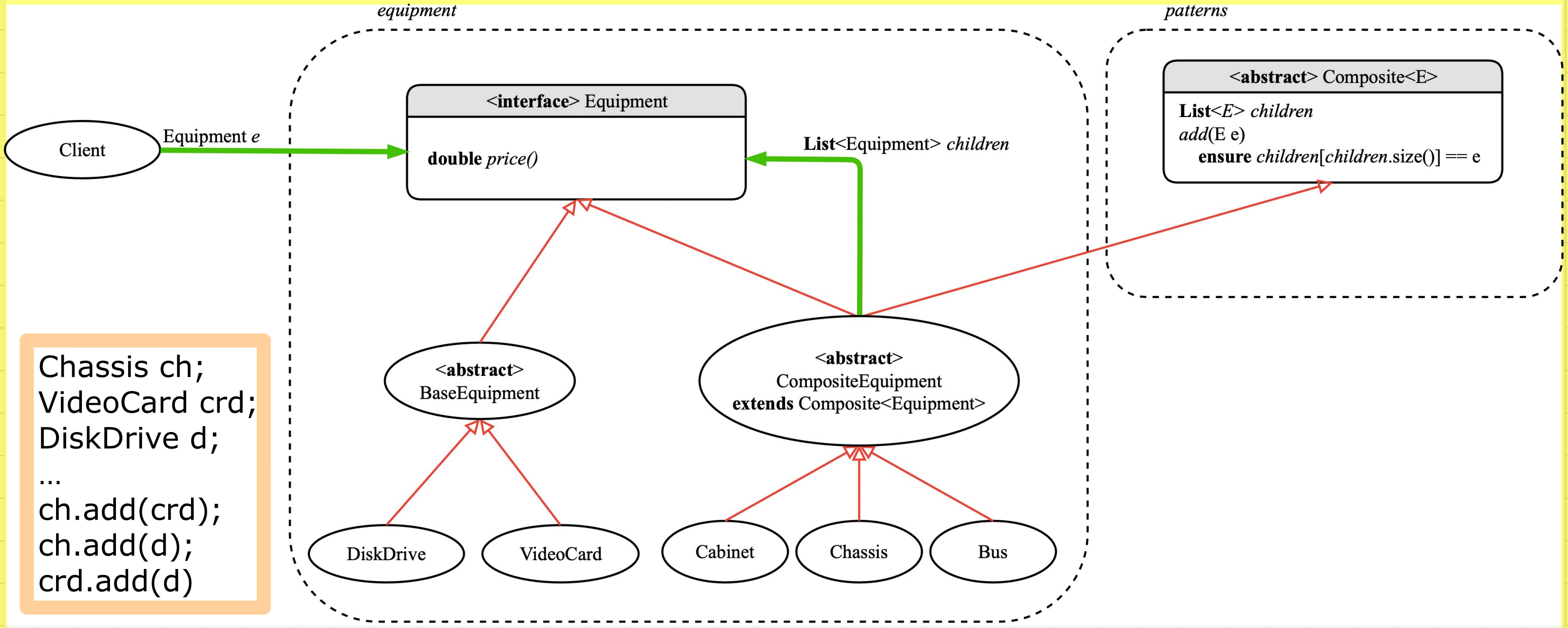
```
abstract void doSomething();
```

Composite Pattern: Architecture



```
Chassis ch;
VideoCard crd;
DiskDrive d;
…
ch.add(crd);
ch.add(d);
crd.add(d)
```

Composite Pattern: Architecture

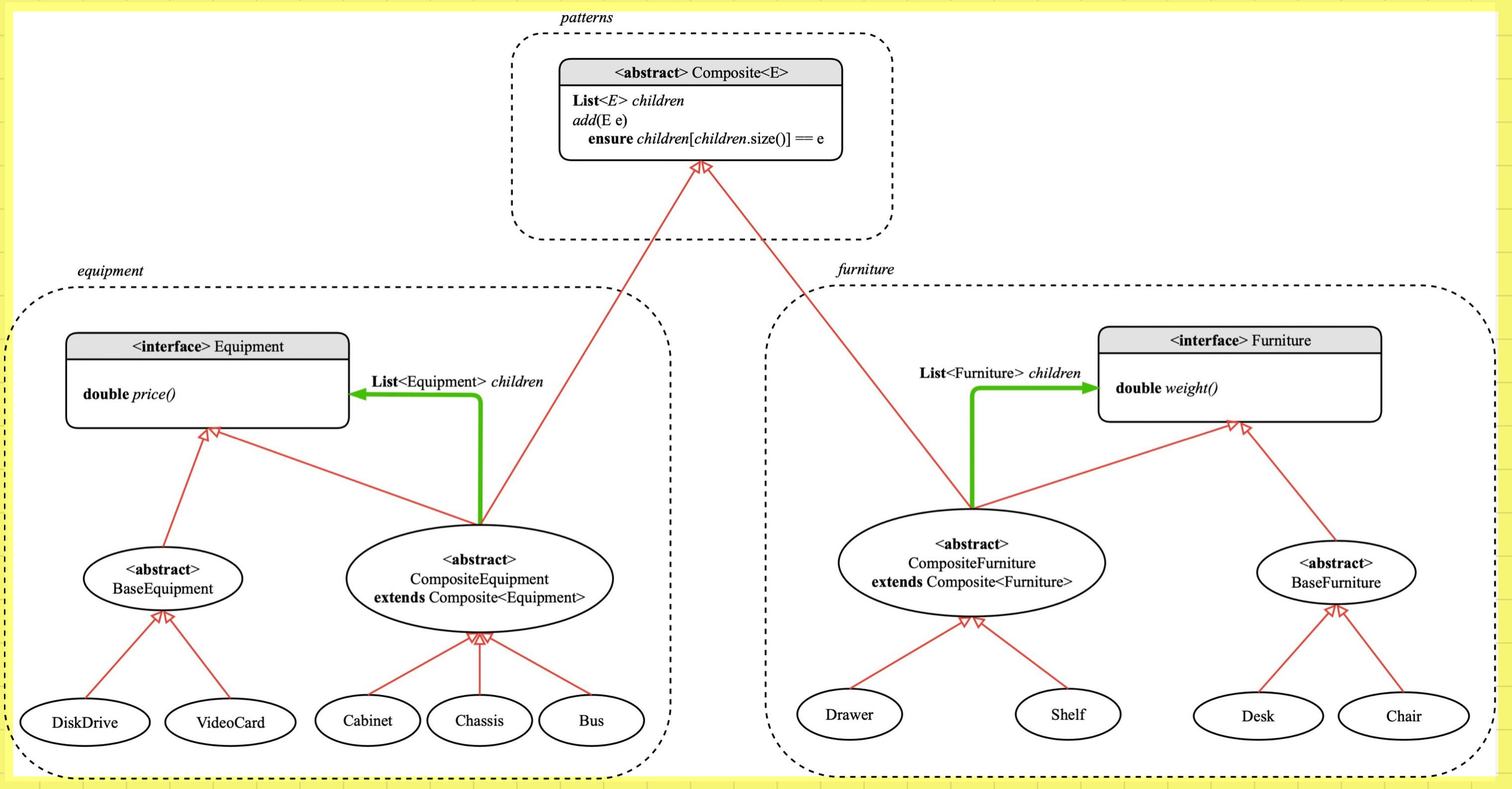


```
Chassis ch;
VideoCard crd;
DiskDrive d;
…
ch.add(crd);
ch.add(d);
crd.add(d)
```

Why is **Composite** a separate, generic class?

Composite Pattern: Architecture

Composite class is **reusable** by instances of the **composite** pattern.



Composite Pattern: Implementation

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

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

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

```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;
  }
}
```

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

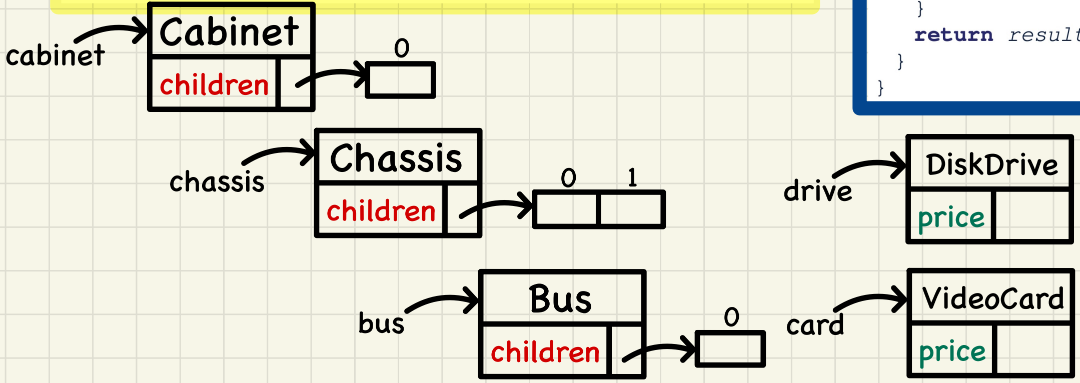
```java
public class Chassis extends CompositeEquipment {
  public Chassis(String name) {
    super(name);
  }
}
```

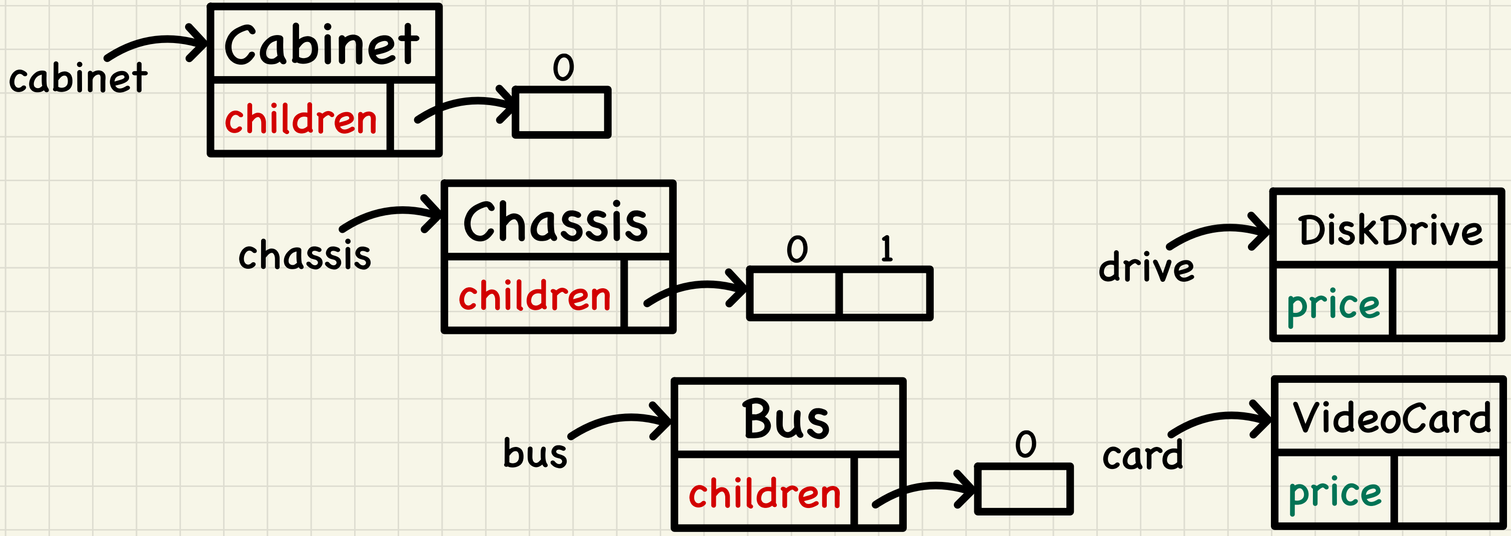
Composite Pattern: Testing

```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);
}
```

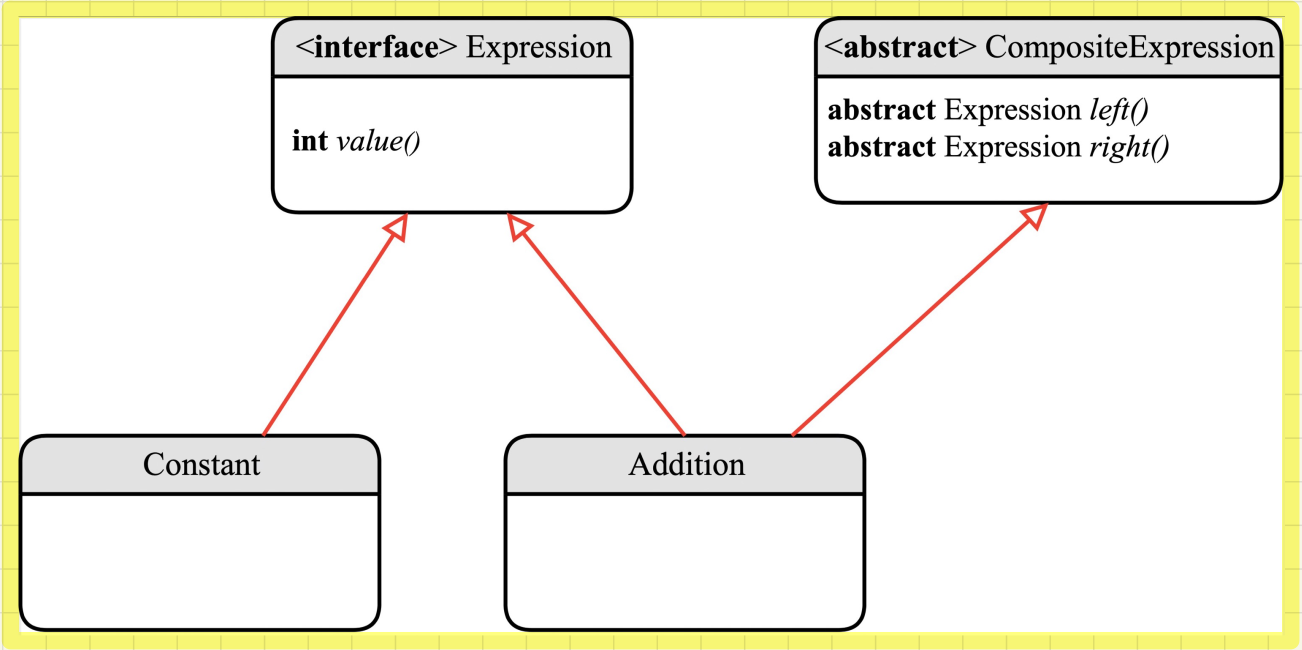
```java
public abstract class BaseEquipment implements Equipment {
  private String name;
  private double price;
  public BaseEquipment(String name, double price) {
    this.name = name; this.price = price;
  }
  public String name() { return this.name; }
  public double price() { return this.price;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

```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;
  }
}
```




cabinet
Cabinet
children
0
chassis
Chassis
children
0
1
drive
DiskDrive
price
bus
Bus
children
0
card
VideoCard
price

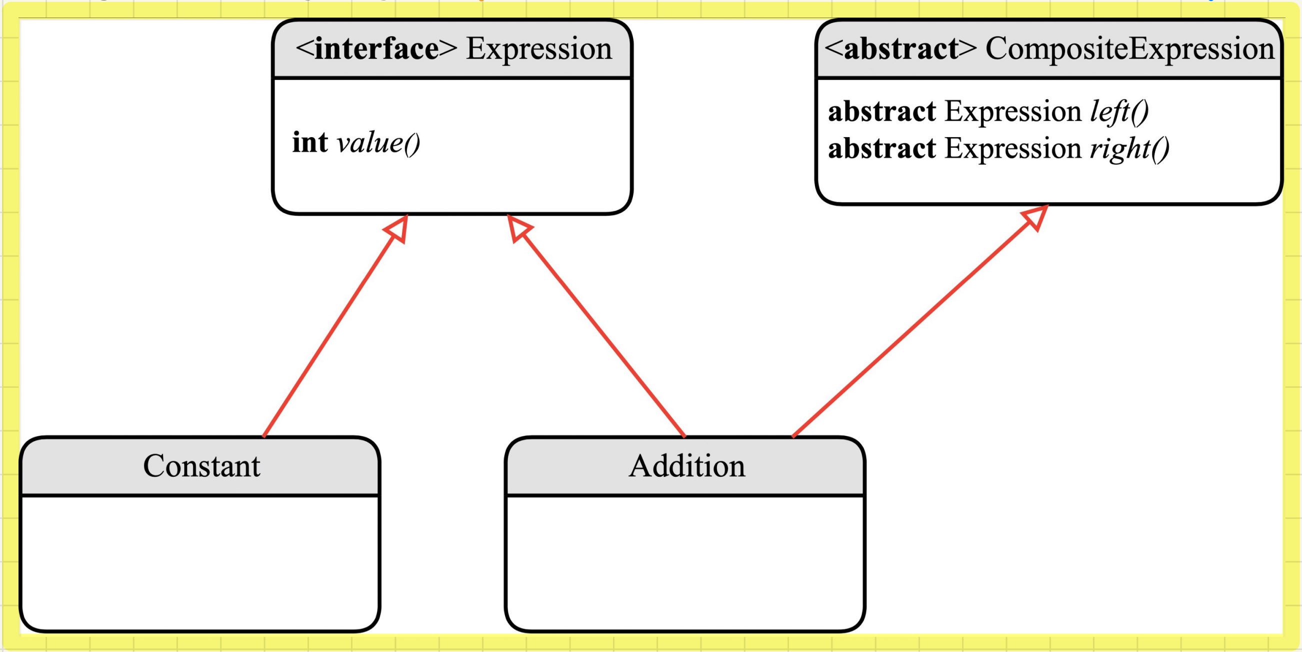
Design of Language Structure: Composite Pattern


<interface> Expression
int value()
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
Addition

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?

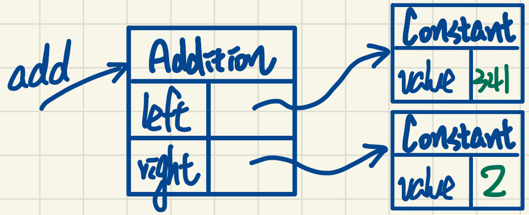


<interface> Expression
int value()
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
Addition

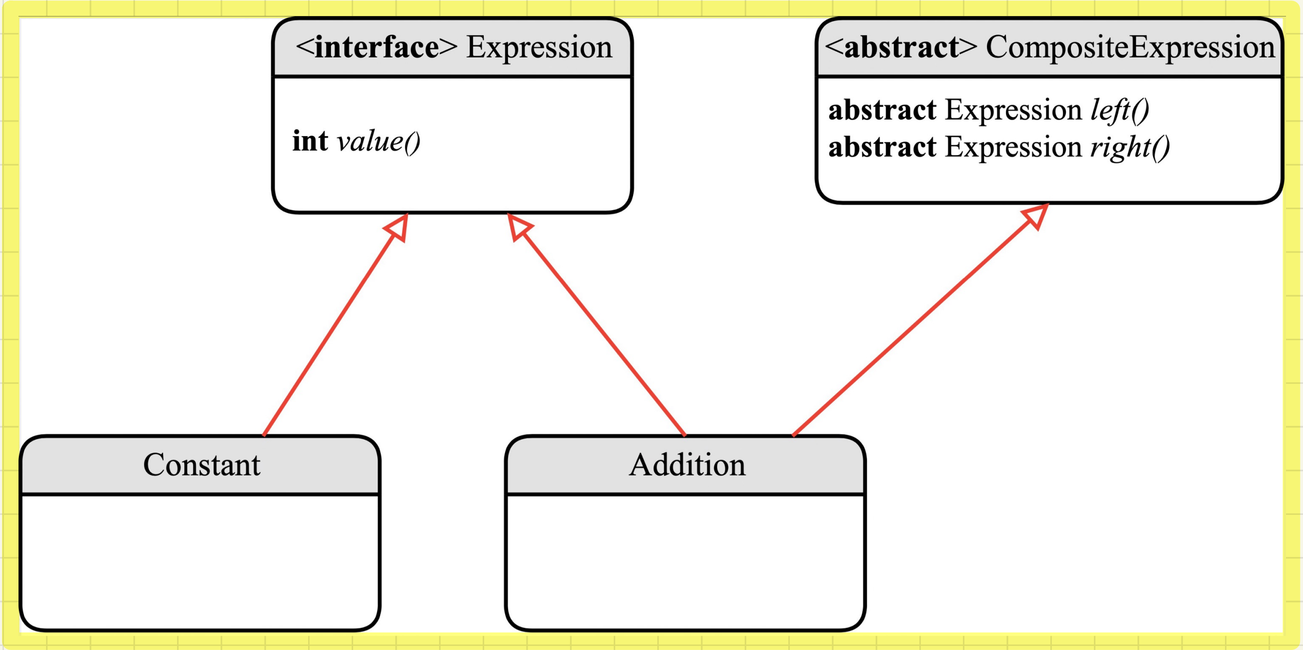
Structure

- evaluate
- print_prefix
- print_postfix
- type_check

Operations



Design of a Language Application: Open-Closed Principle



<interface> Expression
int value()
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
Addition

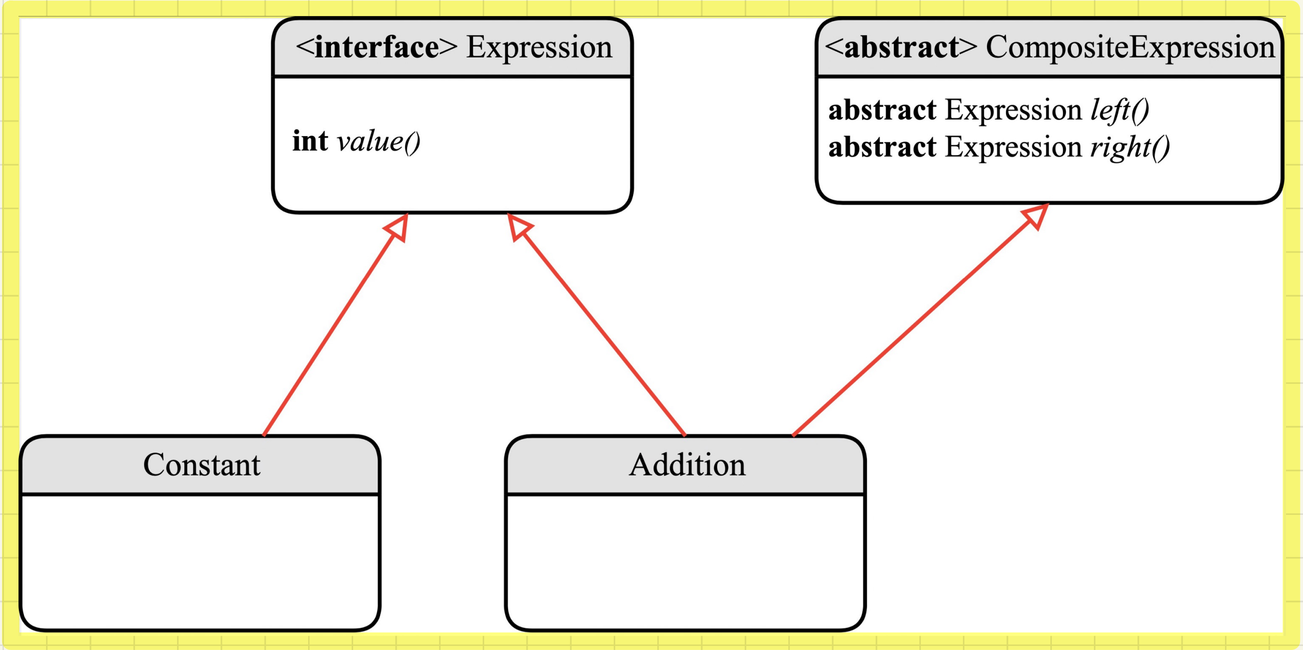
Structure

evaluate
print_prefix
print_postfix
type_check

Operations

	Structure	Operations
Alternative 1	Open	Closed
Alternative 2	Closed	Open

Design of a Language Application: Open-Closed Principle


<interface> Expression
int value()
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
Addition

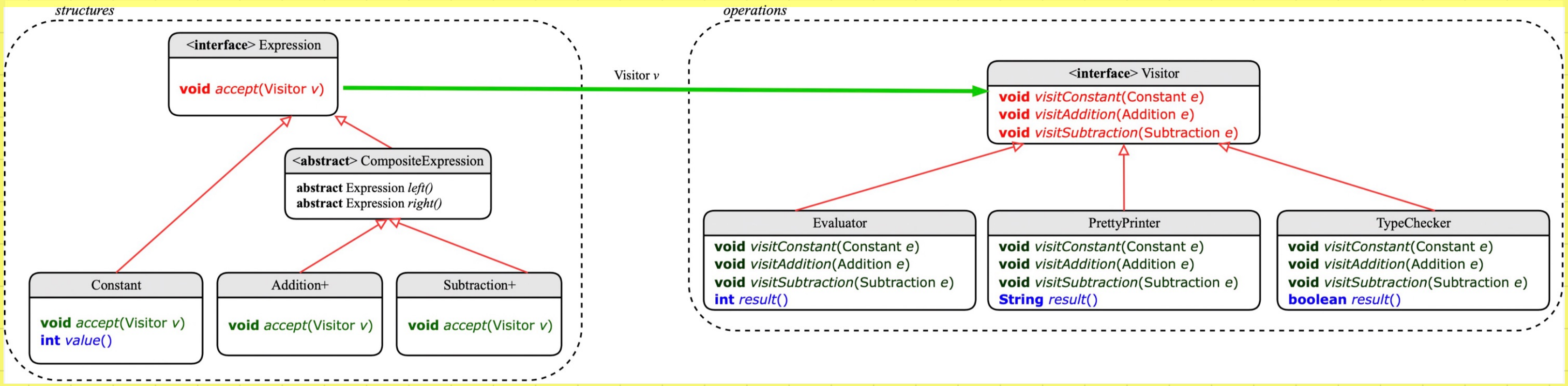
Structure

- evaluate
- print_prefix
- print_postfix
- type_check

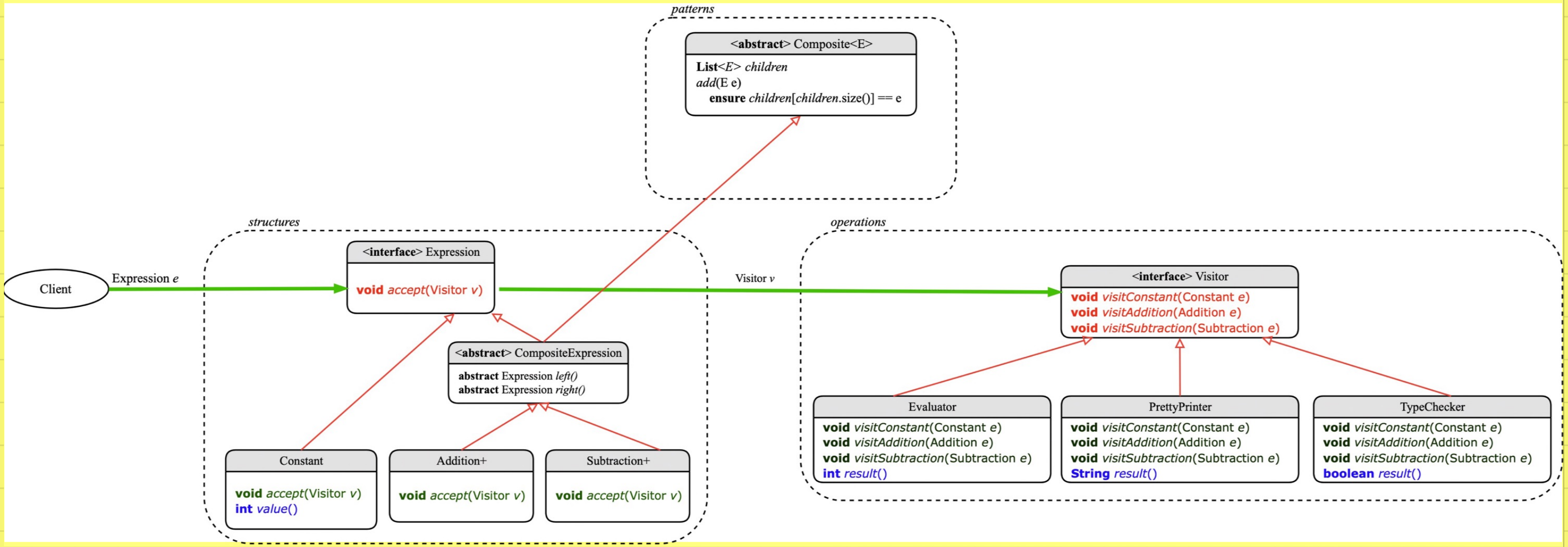
Operations

	Structure	Operations
Alternative 1	Open	Closed
Alternative 2	Closed	Open

Visitor Design Pattern: Architecture


structures
operations
<interface> Expression
void accept(Visitor v)
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
void accept(Visitor v)
int value()
Addition+
void accept(Visitor v)
Subtraction+
void accept(Visitor v)
Visitor v
<interface> Visitor
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
Evaluator
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
int result()
PrettyPrinter
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
String result()
TypeChecker
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
boolean result()

Visitor Design Pattern: Architecture


patterns
<abstract> Composite<E>
List<E> children
add(E e)
ensure children[children.size()] == e
structures
<interface> Expression
void accept(Visitor v)
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
void accept(Visitor v)
int value()
Addition+
void accept(Visitor v)
Subtraction+
void accept(Visitor v)
Client
Expression e
Visitor v
operations
<interface> Visitor
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
Evaluator
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
int result()
PrettyPrinter
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
String result()
TypeChecker
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
boolean result()

```java
@Test
public void test_expression_evaluation() {
  CompositeExpression add;
  Expression c1, c2;
  Visitor v;
  c1 = new Constant(1); c2 = new Constant(2);
  add = new Addition(c1, c2);
  v = new Evaluator();
  add.accept(v);
  assertEquals(3, ((Evaluator) v).result());
}
```

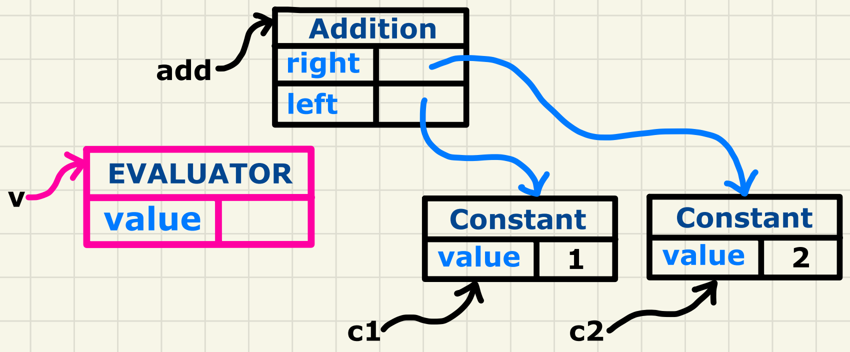
How to Use Visitors

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

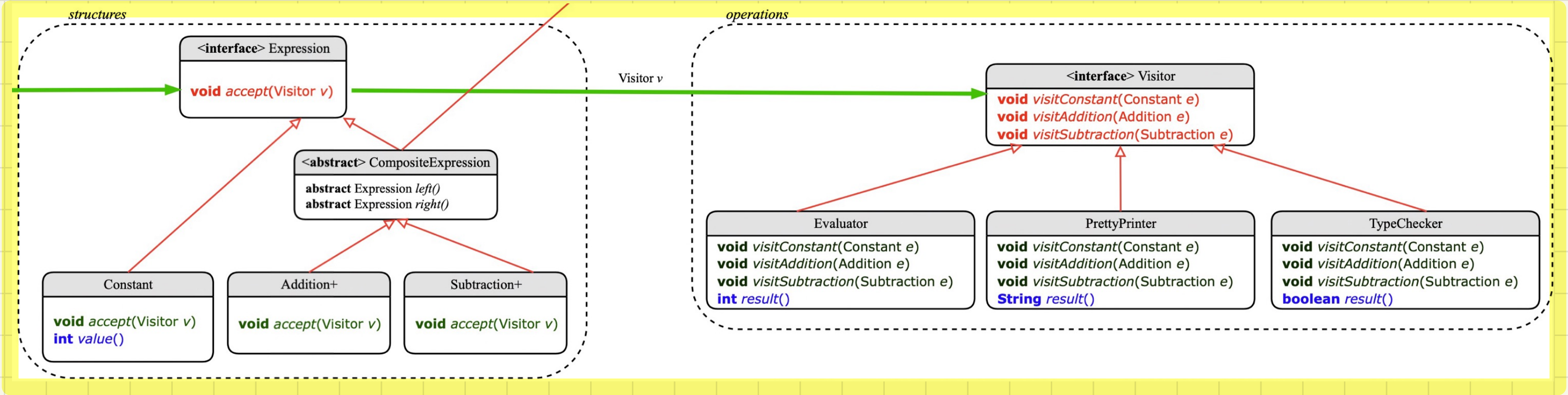
```
public interface Visitor {
  public void visitConstant(Constant e);
  public void visitAddition(Addition e);
  public void visitSubtraction(Subtraction e);
}
```

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

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

```
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();
  }
}
```

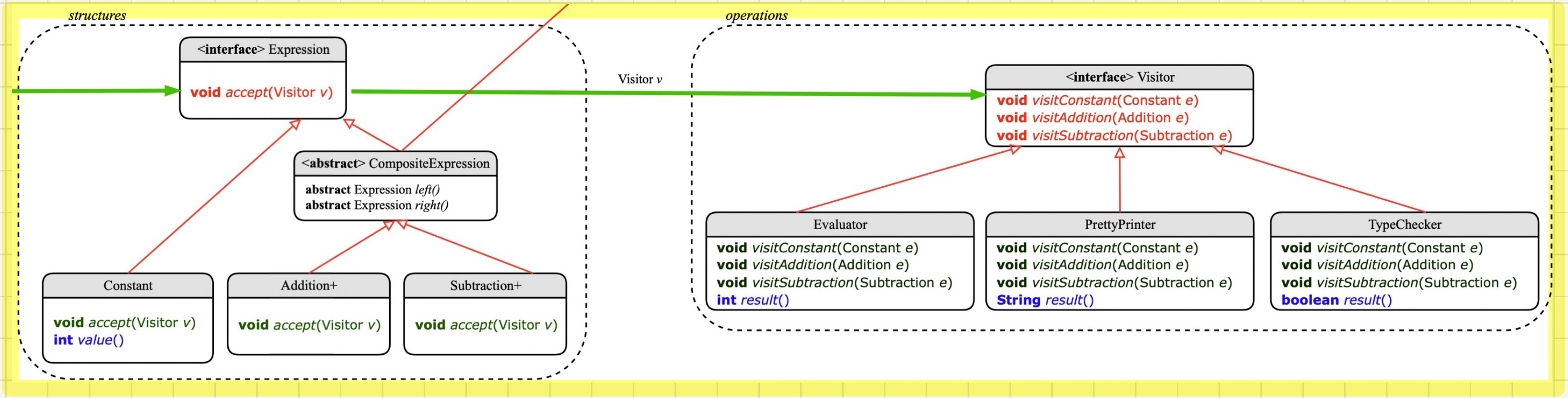
Visitor Pattern: Open-Closed and Single-Choice Principles


structures
operations
<interface> Expression
void accept(Visitor v)
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
void accept(Visitor v)
int value()
Addition+
void accept(Visitor v)
Subtraction+
void accept(Visitor v)
Visitor v
<interface> Visitor
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
Evaluator
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
int result()
PrettyPrinter
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
String result()
TypeChecker
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
boolean result()

What if a new language construct is added?

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

Visitor Pattern: Open-Closed and Single-Choice Principles


structures
operations
<interface> Expression
void accept(Visitor v)
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
void accept(Visitor v)
int value()
Addition+
void accept(Visitor v)
Subtraction+
void accept(Visitor v)
Visitor v
<interface> Visitor
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
Evaluator
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
int result()
PrettyPrinter
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
String result()
TypeChecker
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
boolean result()

What if a new language operation is added?

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