Lecture 13 - Oct. 27

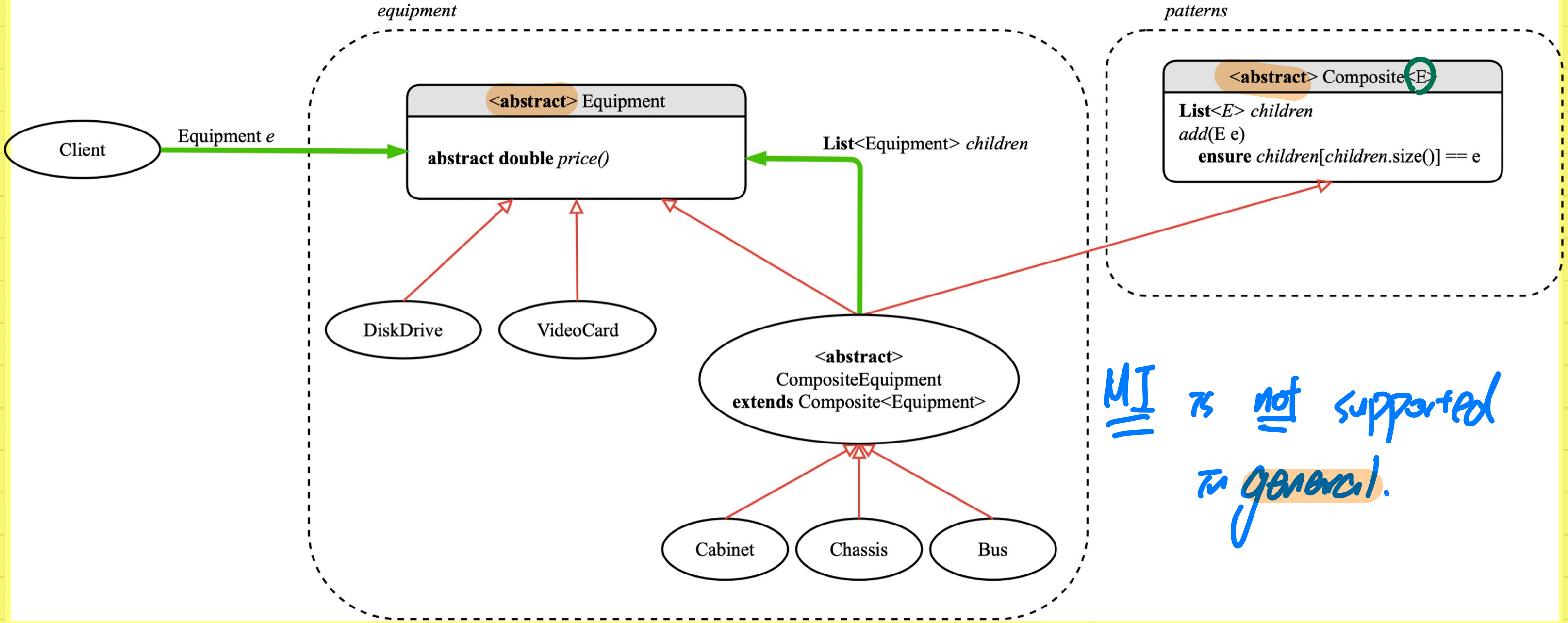
Composite & Visitor

Composite:
Architecture, Implementation, Tests
Visitor:
Architecture, Double Dispatch

Announcements

- Programming Test
  + 2:00pm to 3:20pm on Saturday, October 29
  + Venue: LAS1006 (the large lab)
- Quiz 3
- Project team.txt file due today
- Project Milestone 1

Third Design Attempt



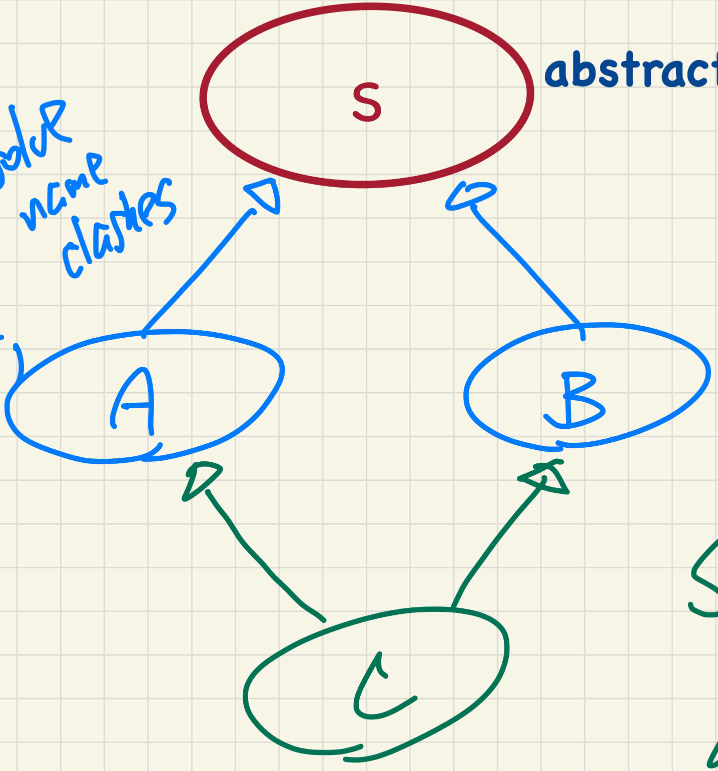
MI is not supported in general.

- $\boxed{\text{abstract class}}$ → a class can extend at most one class (abstract or not)
  - ↳ method: abstract vs. non-abstract
  - ↳ non-static attribute

- $\boxed{\text{interface}}$ → implement multiple interface
  - ↳ all methods are abstract
  - ↳ no non-static attributes
  - ↳ may declare static variable

Multiple Inheritance in Java: Diamond Problem

abstract void doSomething();



Eiffel
↳ MI
↳ rename
name
clashes

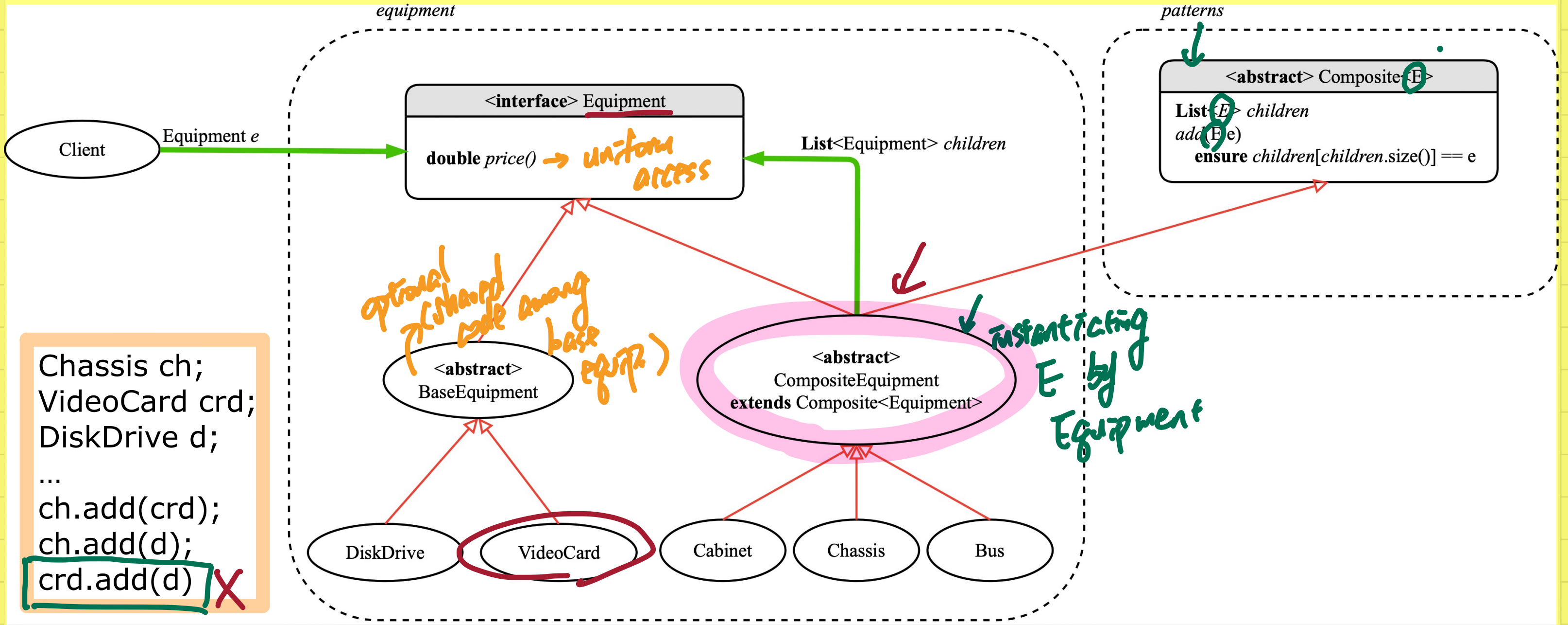
v1
doSomething()

v2.
doSomething()

S obj = new C();

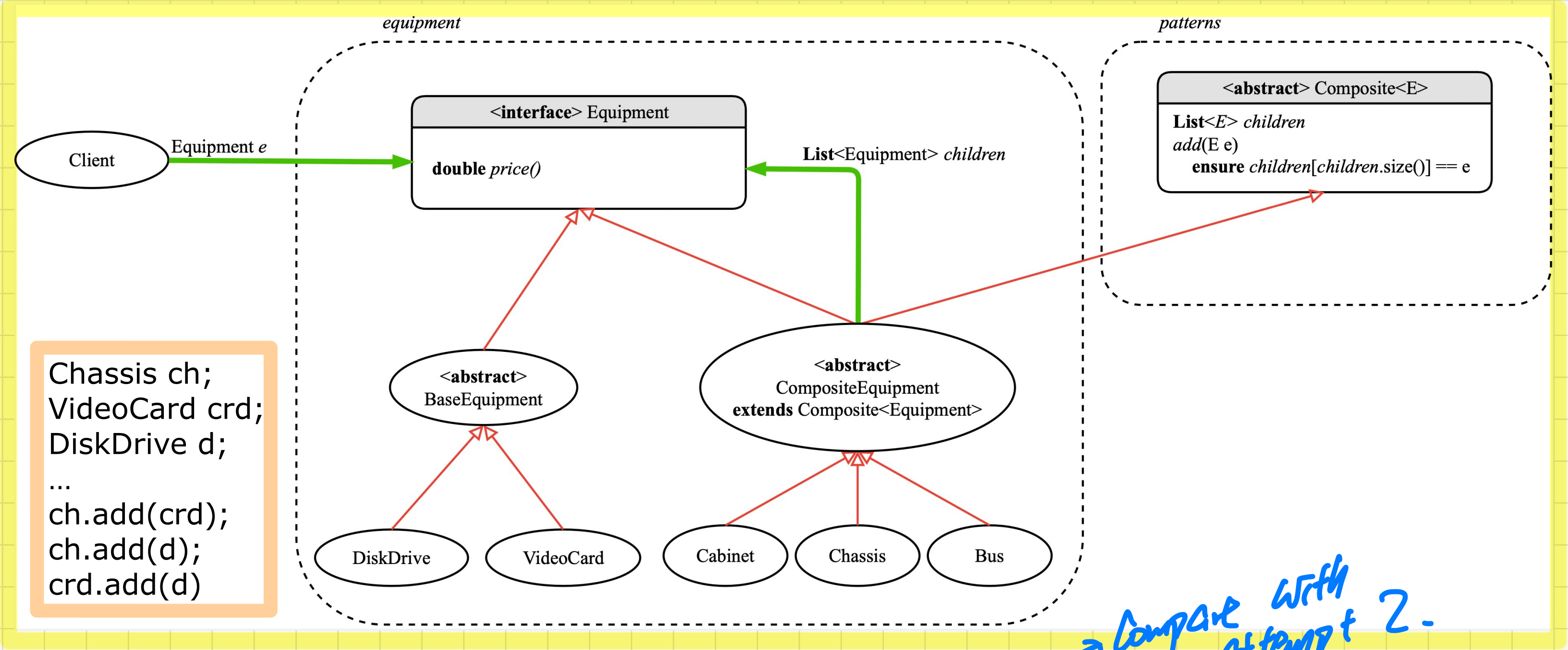
obj.doSomething();

Composite Pattern: Architecture



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

Composite Pattern: Architecture



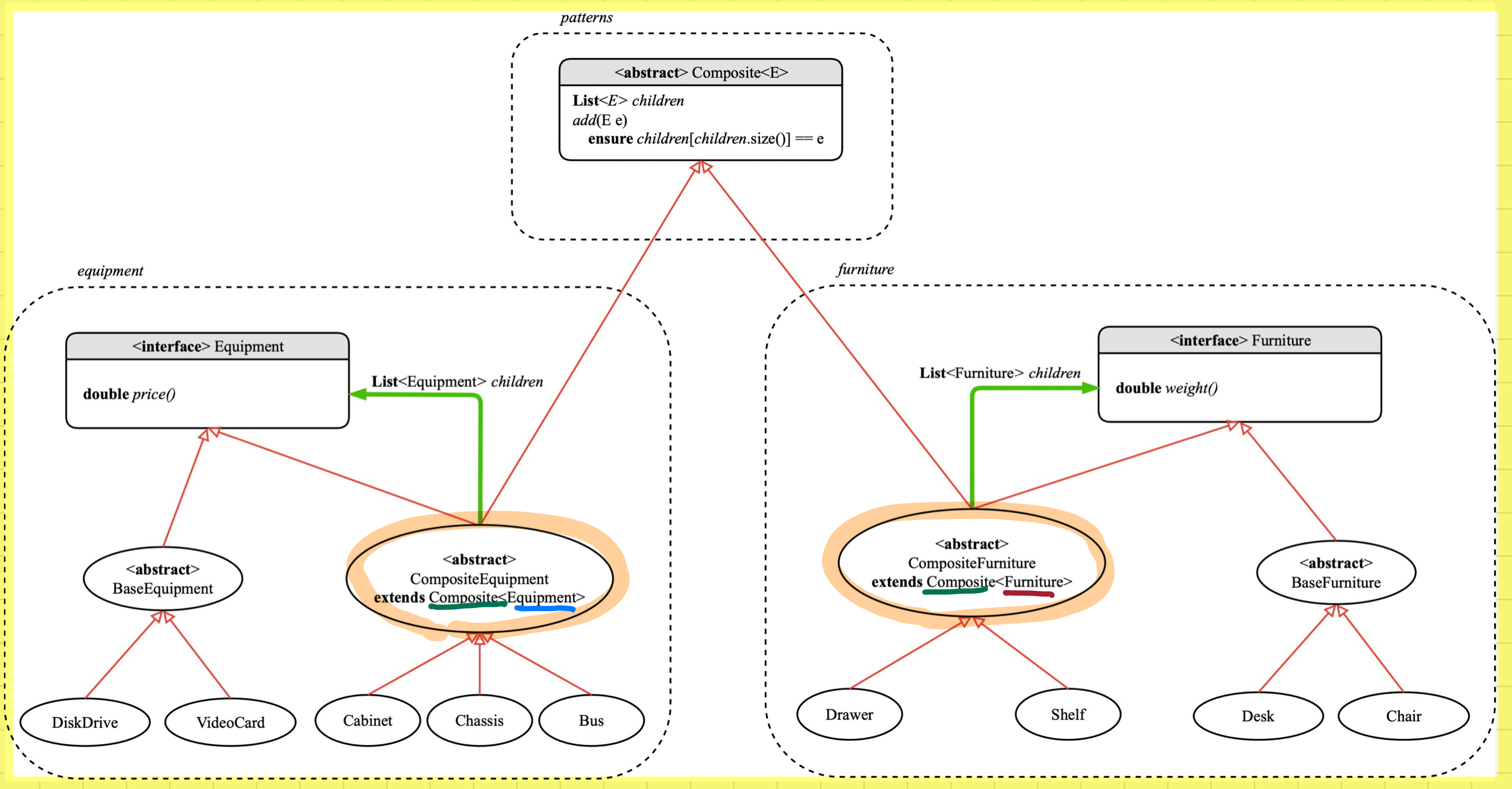
```
Chassis ch;
VideoCard crd;
DiskDrive d;
...
ch.add(crd);
ch.add(d);
crd.add(d)
```

Compare with attempt 2.

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 */
}
```

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; }
}
```

access!

```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;
  }
}
```

DT can be either Base or Composite

uniform access

```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);
}
```

500

200

~~chassis.children[1].price()~~

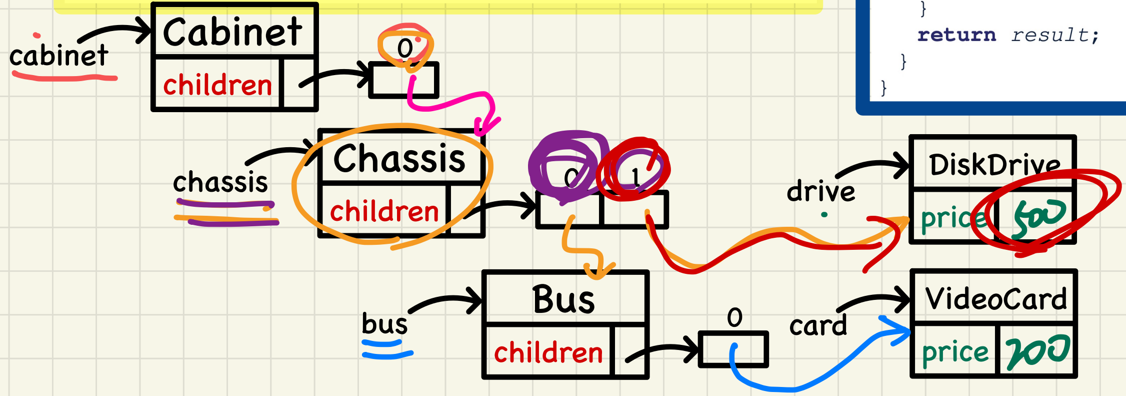
~~chassis.children[0].price()~~

cabinet.c[0].price()

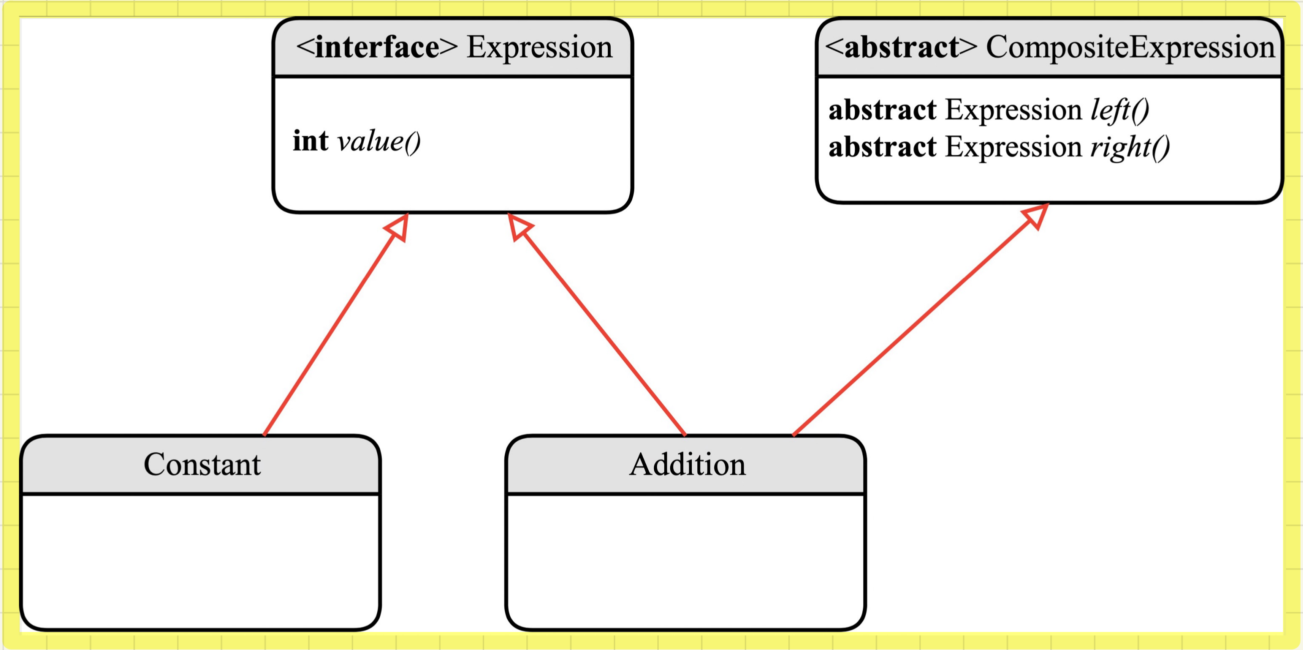
cabinet.price()

```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;
  }
}
```



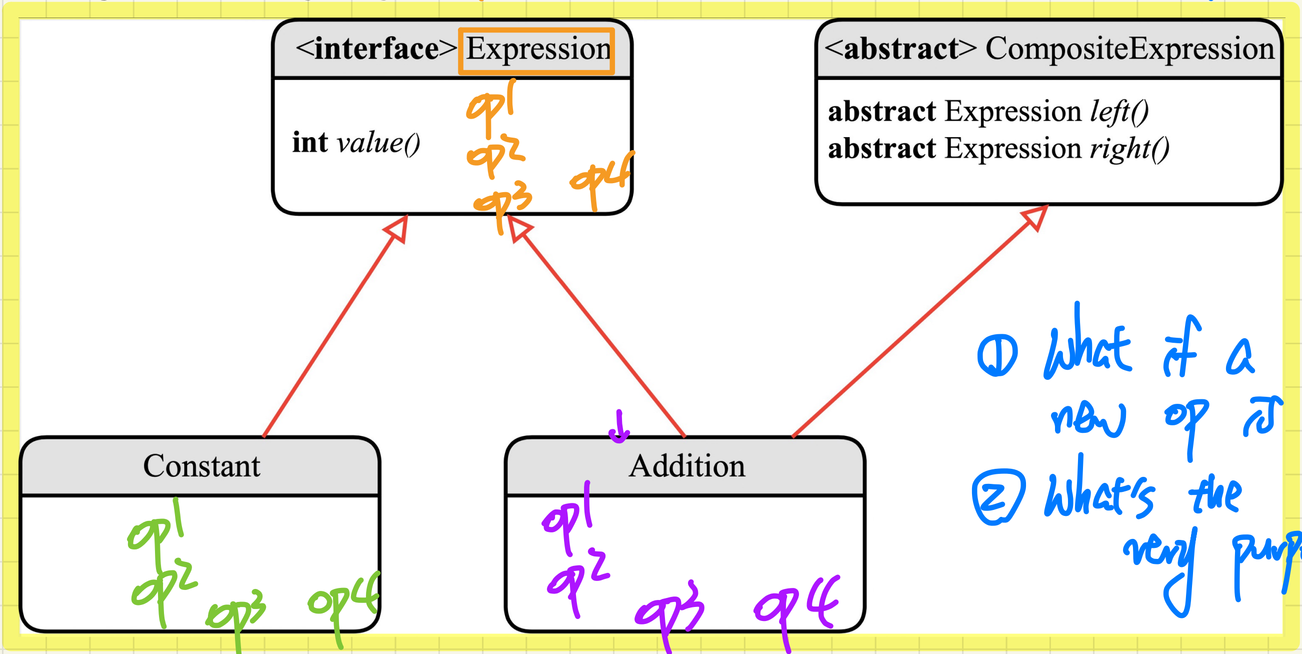
Design of Language Structure: Composite Pattern


<**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?



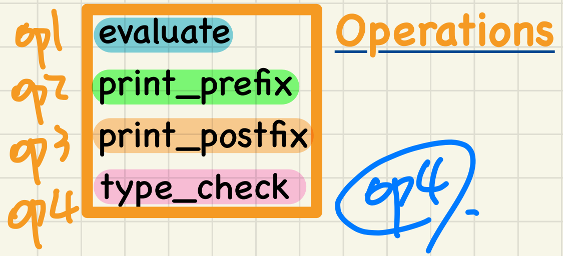
Structure

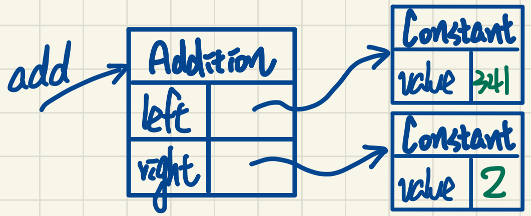
① What if a new op is needed?

② What's the very purpose of a class?

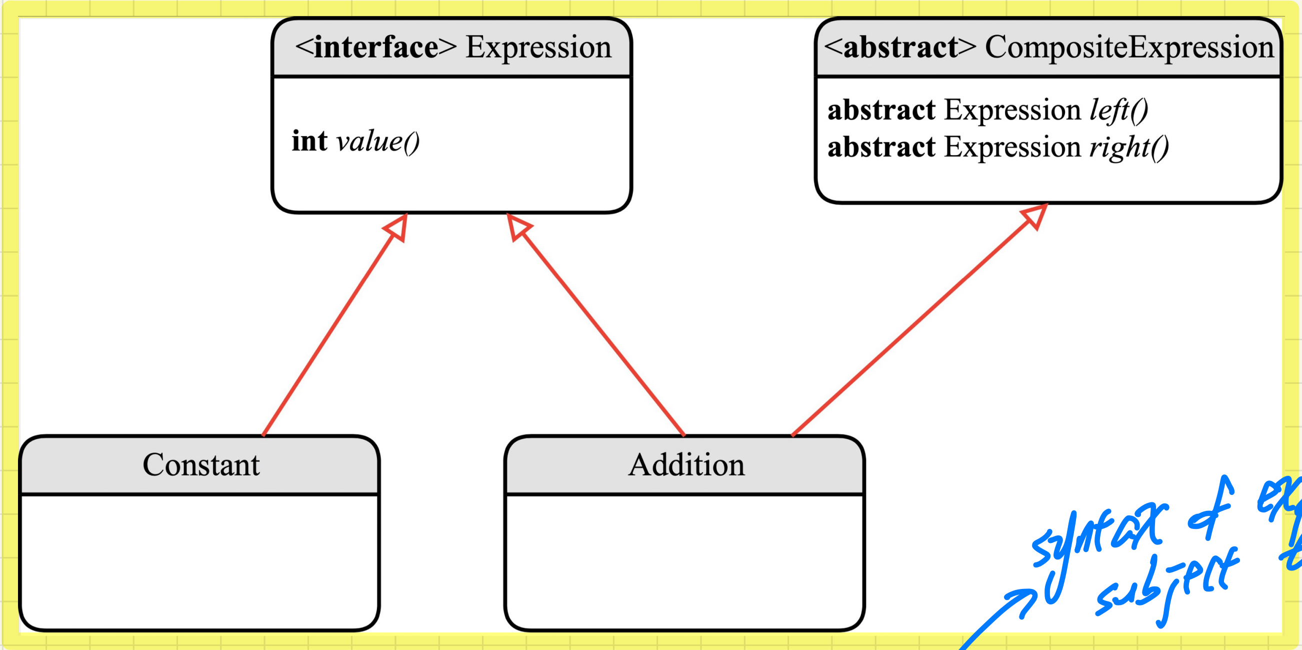
(superman class)

modularity





Design of a Language Application: Open-Closed Principle



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

Structure

syntax of exp. subject to changes

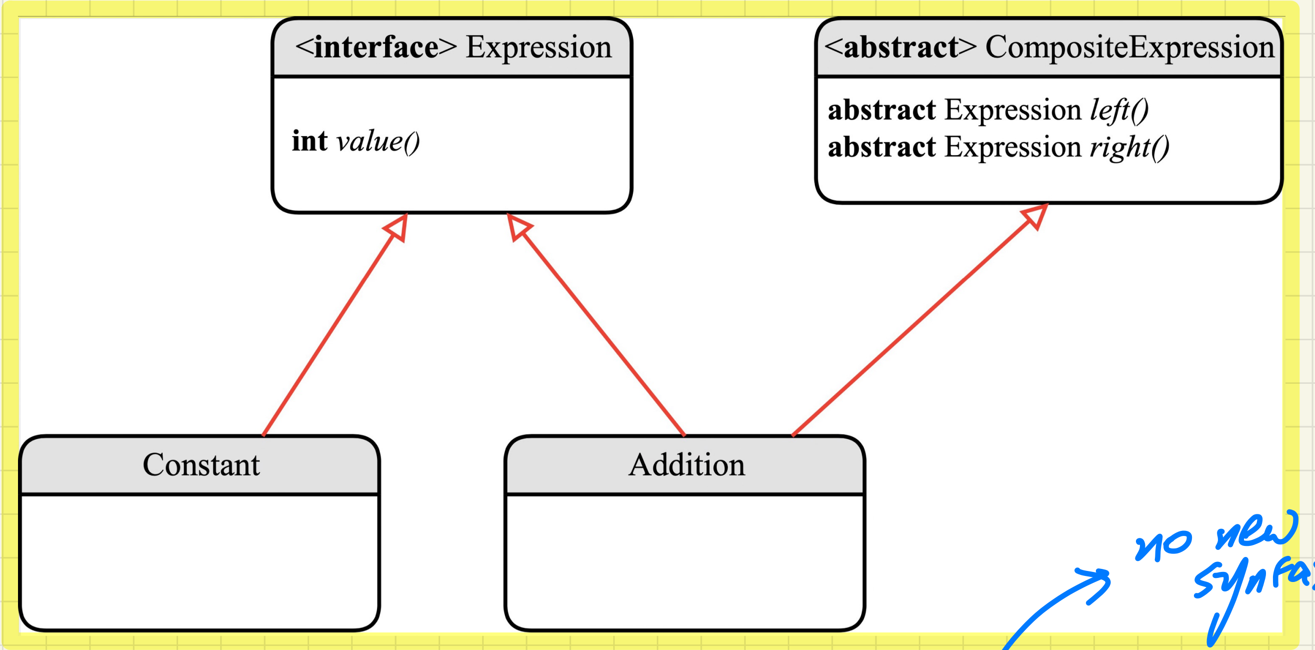
evaluate
print_prefix
print_postfix
type_check

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



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

Structure

evaluate
print_prefix
print_postfix
type_check

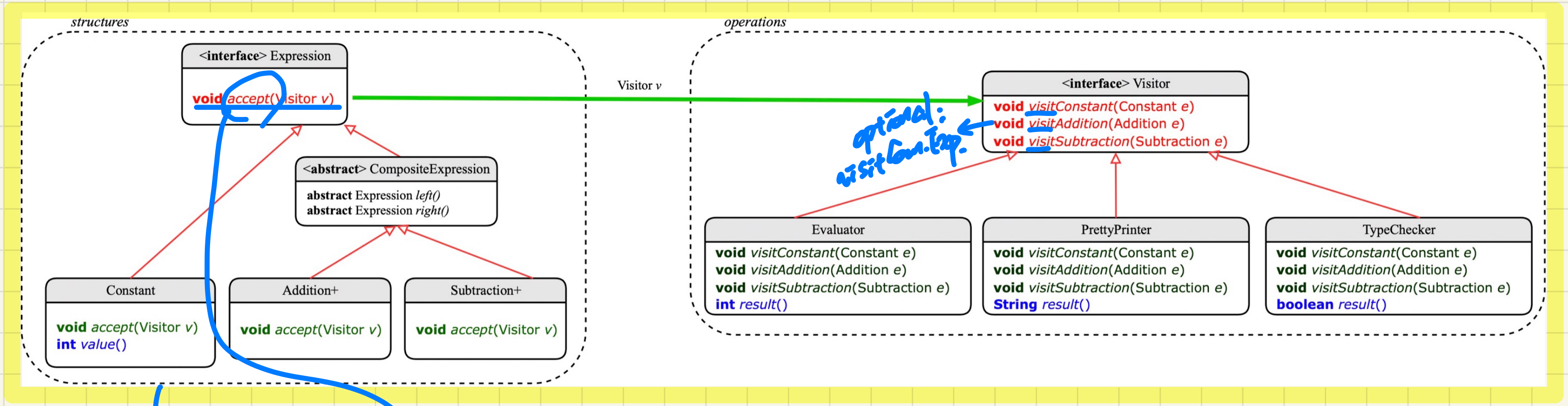
Operations

	Structure	Operations
Alternative 1	Open	Closed
Alternative 2	Closed	Open

no new syntax

keep adding new ops.

Visitor Design Pattern: Architecture

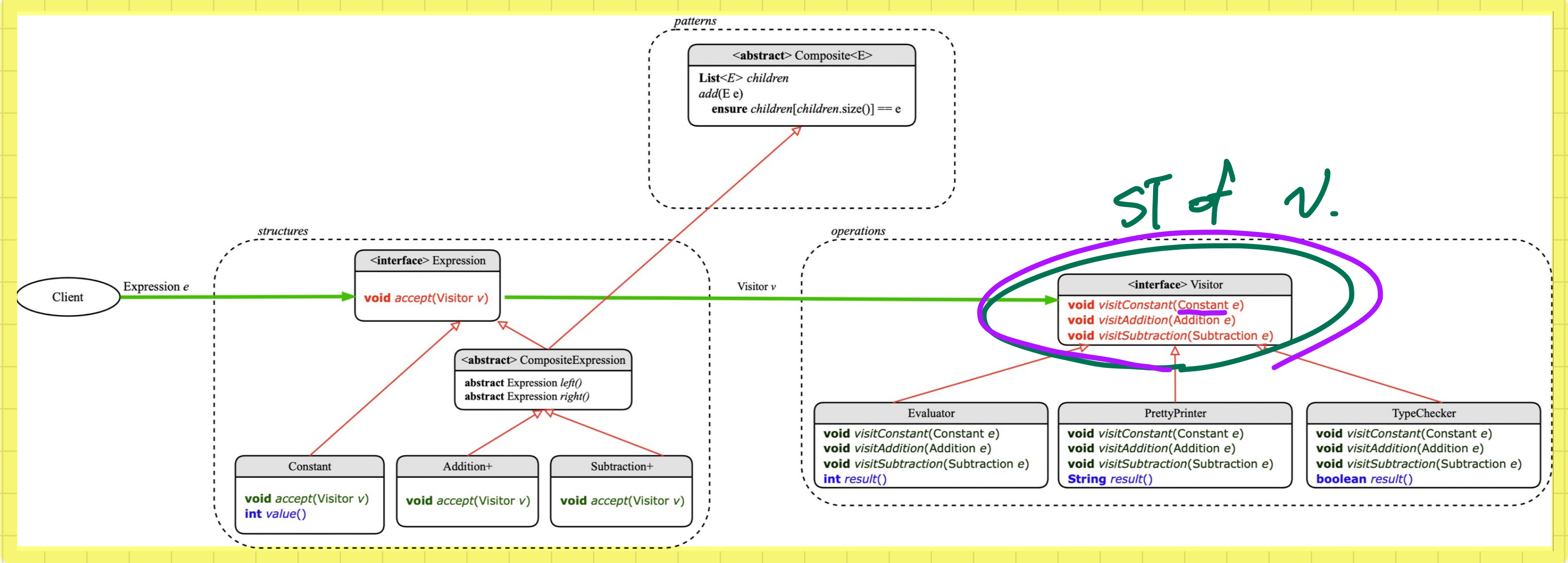

structures
<interface> Expression
void accept(Visitor v)
<abstract> CompositeExpression
abstract Expression left()
abstract Expression right()
Constant
void accept(Visitor v)
int value()
Addition+
void accept(Visitor v)
Subtraction+
void accept(Visitor v)
Visitor v
operations
<interface> Visitor
void visitConstant(Constant e)
void visitAddition(Addition e)
void visitSubtraction(Subtraction e)
optional: visitCom.Exp.
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()

Composite.

Expression e = $\overset{DT.}{\boxed{}}$

e.accept($\overset{\text{visitor}}{\underline{\qquad\qquad}}$)

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()
ST of v.

How to Use Visitors

```
@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());
}
```

static type

dynamic type $\boxed{1+2}$.

root of AST to start processing

Can I write v.result()? v is Visitors

→ ST of v which does not support result()

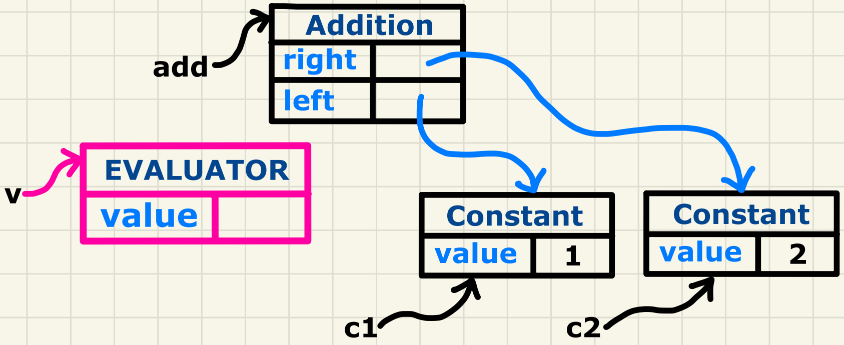
Visitor Design Pattern: Implementation

```
 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

DT



Tracing add.accept(v)

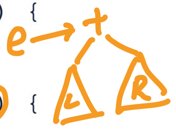
Double Dispatch

DT

```
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 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();
  }
}
```



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