Composite & Visitor Design Patterns

EECS4302 A:
Compilers and Interpreters
Summer 2025

CHEN-WEI WANG

1. Motivating Problem: *Recursive* Systems
2. Three Design Attempts
3. Inheritance: *Abstract Class* vs. *Interface*
4. Fourth Design Attempt: *Composite Design Pattern*
5. Implementing and Testing the Composite Design Pattern

Backgroundn Readings

You may want to review the advanced OOP concepts from
EECS2030 (https://www.eecs.yorku.ca/~wangcw/
teaching/lectures/index.html#EECS2030_F21):

- Inheritance   [Weeks 7, 8, 9]
  - Static Type vs. Dynamic Types
  - Polymorphic Variable Assignment
  - Polymorphic Arrays
  - Dynamic Binding

- Genericity   [Weeks 10, 11]

- Many manufactured systems, such as computer systems or stereo systems, are composed of ***individual components*** and ***sub-systems*** that contain components.

    e.g., A computer system is composed of:
    - <u>Base</u> equipment (*hard drives*, *cd-rom drives*)

        e.g., Each *drive* has **properties**: e.g., power consumption and cost.
    - <u>Composite</u> equipment such as *cabinets*, *busses*, and *chassis*

        e.g., Each *cabinet* contains various types of *chassis*, each of which containing components (*hard-drive*, *power-supply*) and *busses* that contain *cards*.

- Design a system that will allow us to easily `build` systems and `compute` their <u>aggregate</u> cost and power consumption.
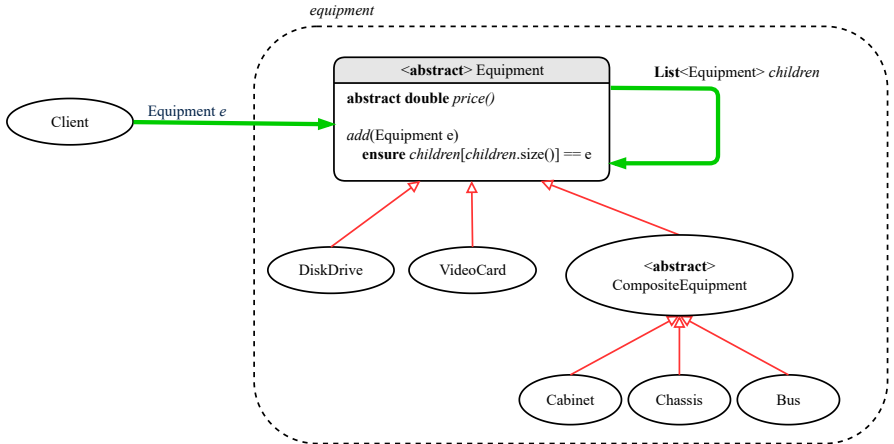
Motivating Problem (2)

Design of *hierarchies* represented in *tree structures*

Challenge : There are *base* and *recursive* modelling artifacts.

equipment

<**abstract**> Equipment

abstract double *price()*

add(Equipment e)
    ensure *children*[*children*.size()] == e

List<Equipment> *children*

Client — Equipment *e* →

DiskDrive

VideoCard

<**abstract**>
CompositeEquipment
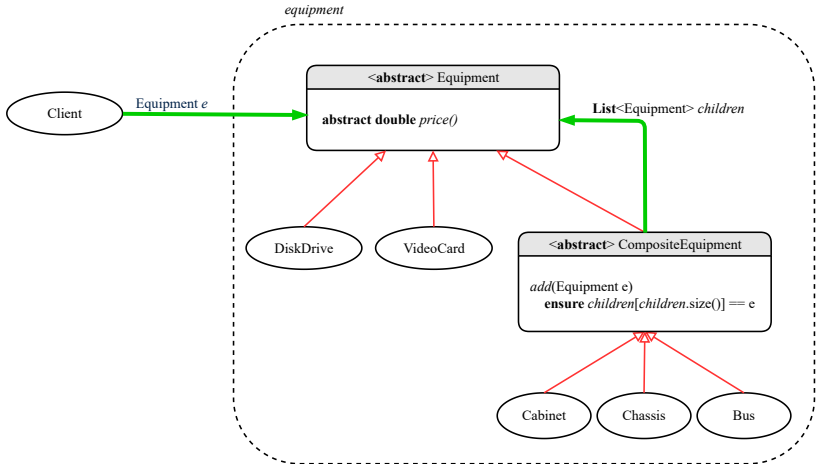
Cabinet

Chassis

Bus

Java List API

Q: Any flaw of this first design?

A: Two "composite" features defined at the Equipment level:
- List<Equipment> children
- add(Equipment child)

⇒ Inherited to each *base* equipment (e.g., DiskDrive), for which such features are **not** applicable.

Design Attempt 2: Flaw?
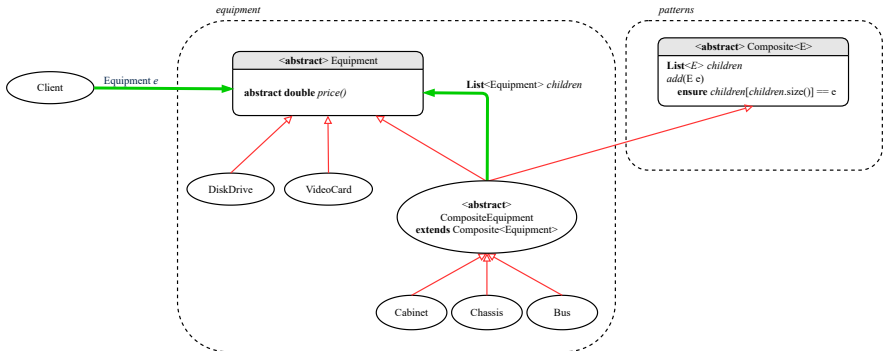
Q: Any flaw of this second design?

A: Two "composite" features defined at the `Composite` level:

- `List<Equipment> children`
- `add(Equipment child)`

⇒ Multiple *types* of the composite (e.g., equipment, furniture) cause duplicates of the `Composite` class.

⇒ Use a *generic (type) parameter* to *abstract* away the *concrete* type of any potential composite.

Q: Any flaw of this third design?

A: It does **not** compile:

Java does not support ***multiple inheritance***!

○ See: https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html

○ A class may inherit from <u>at most one</u> class (**abstract** or not).
  Rationale. *MI* results in name clashes

  [a.k.a. the ***Diamond Problem***].

○ However, a class may implement <u>multiple</u> ***interfaces***.

  [workaround for implementation]
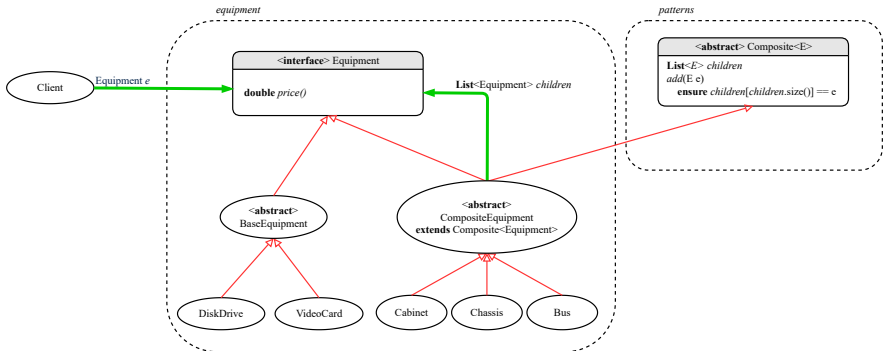
The Composite Pattern: Instantiations
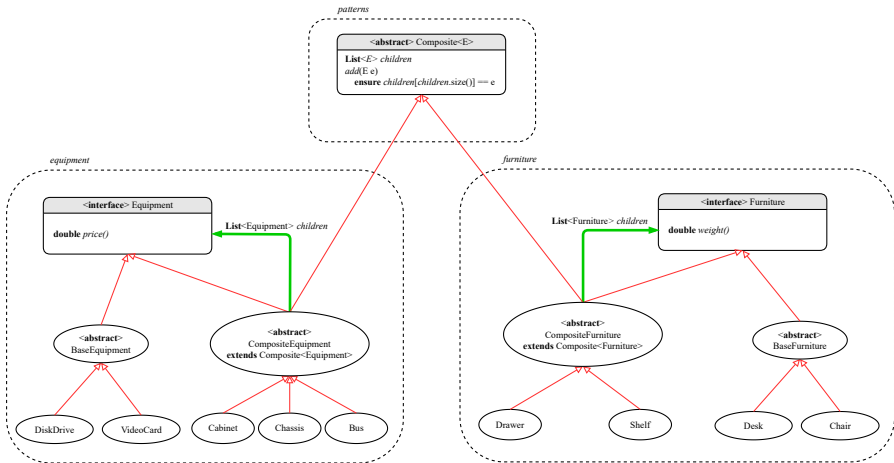
```
public interface Equipment {
 public String name();
 public double price(); /* uniform access */
}
```

```
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; }
}
```

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

```java
import java.util.List;

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

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

```java
import java.util.ArrayList;

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;
 }
}
```

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

```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);
}
```
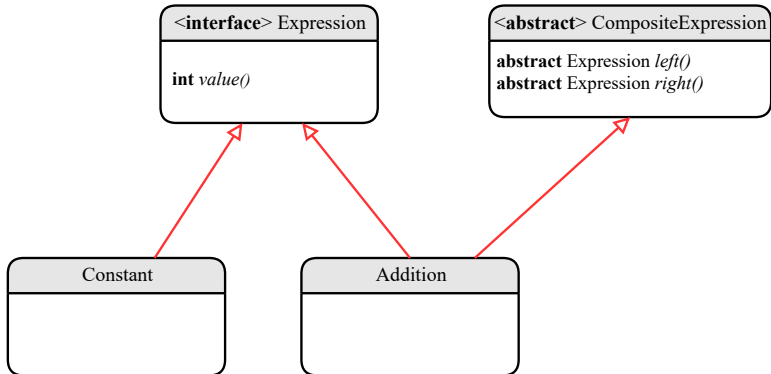
Summary: The Composite Pattern

- **Design** : Categorize into *base* artifacts or *recursive* artifacts.

- **Programming** :
  Build the *tree structure* representing some *hierarchy*.

- **Runtime** :
  Allow clients to treat *base* objects (leafs) and *recursive* compositions (nodes) *uniformly* (e.g., `price()`).

  ⇒ *Polymorphism* : *leafs* and *nodes* are "substitutable".

  ⇒ *Dynamic Binding* : Different versions of the same operation is applied on *base objects* and *composite objects*.
  e.g., Given **Equipment** e :

  ○ `e.price()` may return the unit price, e.g., of a *DiskDrive*.
  ○ `e.price()` may sum prices, e.g., of a *Chassis*' containing equipment.

Learning Objectives

1. Motivating Problem: ***Processing*** Recursive Systems

2. First Design Attempt: Cohesion & Single-Choice Principle?

3. Design Principles:
- *Cohesion*
- *Single Choice* Principle
- *Open-Closed* Principle

4. Second Design Attempt: ***Visitor Design Pattern***
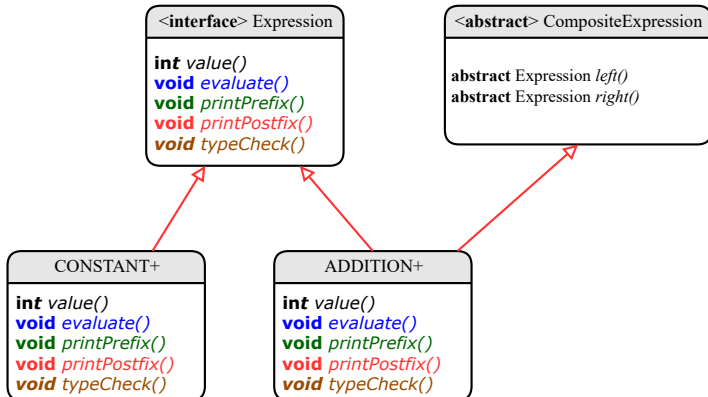
5. Implementing and Testing the Visitor Design Pattern

Based on the *composite pattern* you learned, design classes to model ***structures*** of arithmetic expressions (e.g., *341*, *2*, *341 + 2*).

Motivating Problem (2)

Extend the *composite pattern* to support ***operations*** such as `evaluate`, pretty printing (`print_prefix`, `print_postfix`), and `type_check`.


```
<interface> Expression
```
in*t value()*
void *evaluate()*
void *printPrefix()*
void *printPostfix()*
void *typeCheck()*

```
<abstract> CompositeExpression
```
abstract Expression *left()*
abstract Expression *right()*

```
CONSTANT+
```
in*t value()*
void *evaluate()*
void *printPrefix()*
void *printPostfix()*
void *typeCheck()*

```
ADDITION+
```
in*t value()*
void *evaluate()*
void *printPrefix()*
void *printPostfix()*
void *typeCheck()*

- *Cohesion*:
  - A class/module groups *relevant* features (data & operations).
- *Single Choice Principle* (SCP):
  - When a *change* is needed, there should be *a single place* (or *a minimal number of places*) where you need to make that change.
  - Violation of SCP means that your design contains *redundancies*.

- Distributing **unrelated** *operations* across nodes of the *abstract syntax tree* violates the *single-choice principle*:
    To add/delete/modify an operation
        ⇒ Change of all descendants of `Expression`
- Each node class lacks in *cohesion*:
    A **class** should group *relevant* concepts in a **single** place.
        ⇒ Confusing to mix codes for evaluation, pretty printing, type checking.
        ⇒ Avoid "polluting" the classes with these **unrelated** operations.

Open/Closed Principle

- Software entities (classes, features, etc.) should be ***open*** for ***extension***, but ***closed*** for ***modification***.
  - ⇒ As a system evolves, we:
  - ◦ May add/modify the ***open*** (unstable) part of system.
  - ◦ May **not** add/modify the ***closed*** (stable) part of system.
- e.g., In designing the application of an expression language:
  - ◦ **ALTERNATIVE 1**:
    Syntactic constructs of the language may be ***open***, whereas operations on the language may be ***closed***.
  - ◦ **ALTERNATIVE 2**:
    Syntactic constructs of the language may be ***closed***, whereas operations on the language may be ***open***.

Visitor Pattern

- **Separation of concerns**:
  - Set of language (syntactic) constructs
  - Set of operations

  $\Rightarrow$ Classes from these two sets are `decoupled` and organized into two separate packages.

- **Open-Closed Principle** (OCP): [**ALTERNATIVE 2**]
  - **Closed**, staple part of system: set of language constructs
  - **Open**, unstable part of system: set of operations

  $\Rightarrow$ **OCP** helps us determine if the **Visitor Pattern** is <u>applicable</u>.

  $\Rightarrow$ If it is determined that language constructs are **open** and operations are **closed**, then do **not** use the Visitor Pattern.

Visitor Pattern: Architecture

LASSONDE
SCHOOL OF ENGINEERING
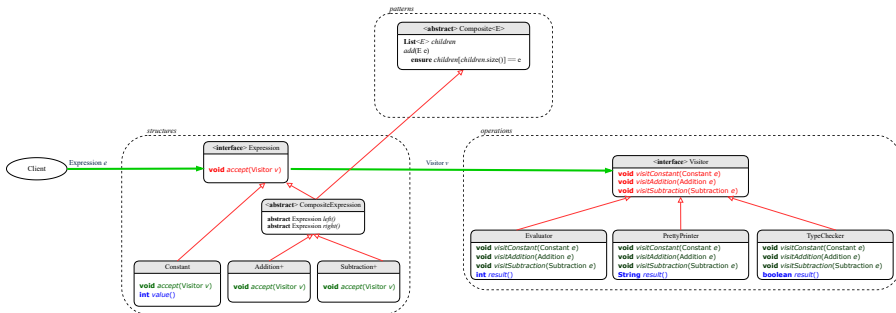
Package *structures*

- Declare ` void accept(Visitor v) ` in <u>abstract</u> class `Expression`.
- Implement `accept` in each of `Expression`'s <u>descendant</u> classes.

```
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);
  }
}
```

Package *operations*

- For each <u>descendant</u> class C of Expression, declare a method header
  | **void** *visitC* (e: *C*) | in the *interface* Visitor.

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

- Each descendant of VISITOR denotes a kind of operation.

```
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();
  }
}
```

Testing the Visitor Pattern

```
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  }
```

Double Dispatch in **Line 9**:

1. **DT** of add is `Addition` ⇒ Call `accept` in `ADDITION`.

   `v.visitAddition(add)`

2. **DT** of v is `Evaluator` ⇒ Call `visitAddition` in `Evaluator`.

   visiting result of `add.left()` + visiting result of `add.right()`

To Use or Not to Use the Visitor Pattern

- In the *visitor pattern*, what kind of *extensions* is easy?
  - Adding a new kind of *operation* element is easy.
    - To introduce a new operation for generating C code, we only need to introduce a new descendant class `CCodeGenerator` of Visitor, then implement how to handle each language element in that class.
    - ⇒ *Single Choice Principle* is <u>satisfied</u>.
- In the *visitor pattern*, what kind of *extensions* is hard?
  - Adding a new kind of *structure* element is hard.
    - After adding a descendant class Multiplcation of Expression, every concrete visitor (i.e., descendant of Visitor) must be amended with a new `visitMultiplication` operation.
    - ⇒ *Single Choice Principle* is <u>violated</u>.
- The applicability of the visitor pattern depends on to what extent the *structure* will change.
  - ⇒ Use visitor if *operations* (applied to structure) change often.
  - ⇒ Do not use visitor if the *structure* changes often.

Index (1)