Composite Pattern

UNSW SYDNEY

Composite Pattern

These lecture notes use material from the reference book "*Head First Design Patterns*".

Composite Pattern: Motivation and Intent

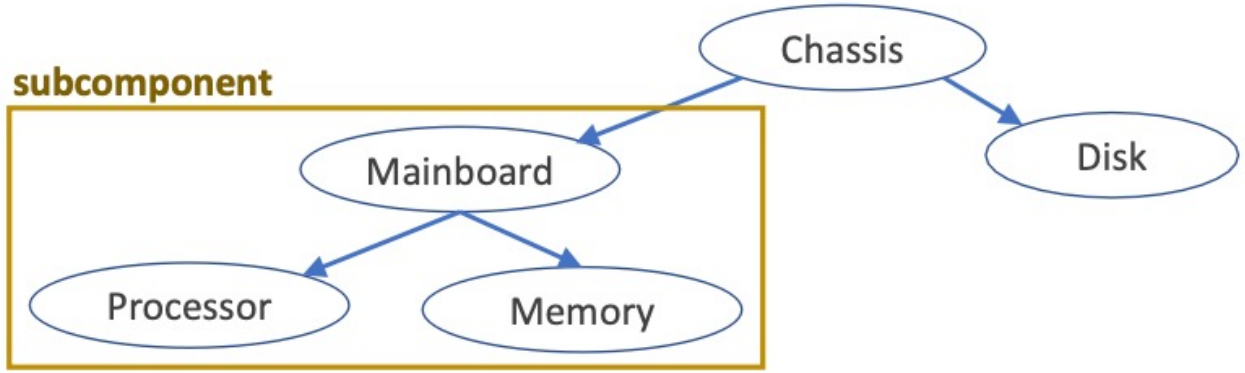
- In OO programming, a composite is an object designed as a composition of one-or-more similar objects (exhibiting similar functionality).

- Aim is to be able to manipulate a single instance of the object just as we would manipulate a group of them. For example,
  - operation to resize a group of Shapes should be **same as** resizing a single Shape.
  - calculating size of a file should be **same as** a directory.

- No discrimination between a Single (leaf) Vs a Composite (group) object.
  - If we discriminate between a single object and a group of object, code will become more complex and therefore, more error prone.

Composite Pattern: More Examples

Calculate the total price of an individual part or a complete subcomponent (consisting of many parts) without having to treat part and subcomponent differently.

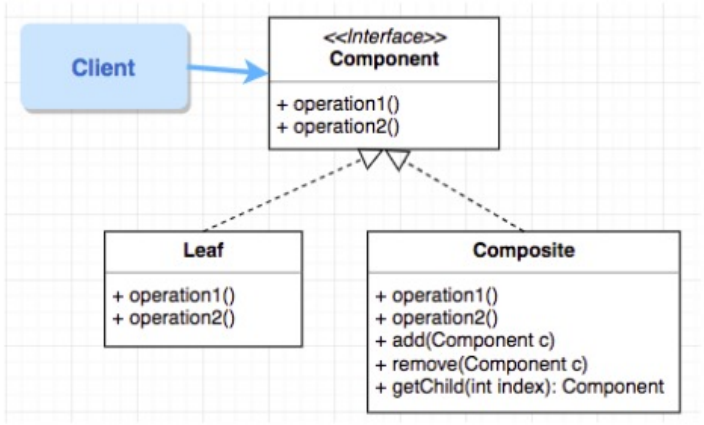


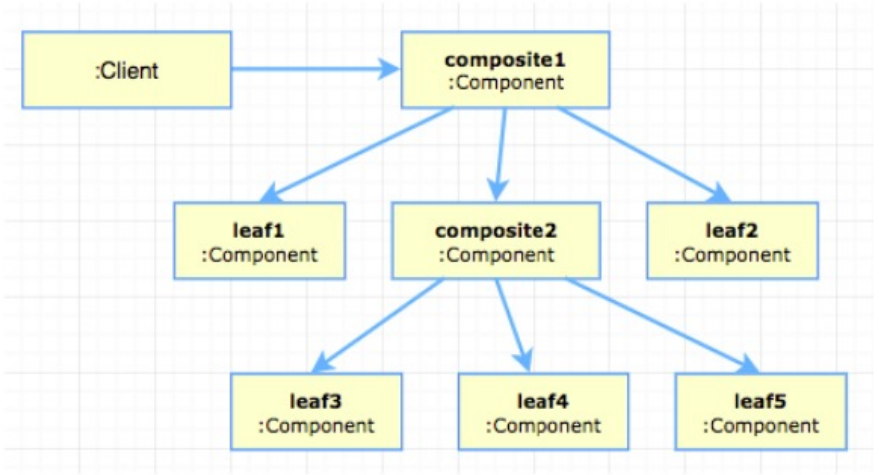
A text document can be organized as part-whole hierarchy consisting of

- characters, pictures, lines, pages, etc. (parts) and
- lines, pages, document, etc. (wholes).

- Display a line, page or the entire document (consisting of many pages) uniformly using the same operation/method.

Composite Pattern: Possible Solution





- Define a unified **`Component`** interface for both **`Leaf`** (*single / part*) objects and **`Composite`** (*Group / whole*) objects.
- A `Composite` stores a **collection of children** components (either `Leaf` and/or `Composite` objects).
- Clients can ignore the differences between compositions of objects and individual objects, this greatly simplifies clients of complex hierarchies and makes them easier to implement, change, test, and reuse.

Composite Pattern: Possible Solution

- *Tree* structures are normally used to represent part-whole hierarchies. A multiway tree structure stores a collection of say `Components` at each node (`children` below), to store `Leaf` objects and `Composite` (subtree) objects.
- A `Leaf` object performs operations directly on the object.
- A `Composite` object performs operations on its **children**, and if required, collects return values and derives the required answers.

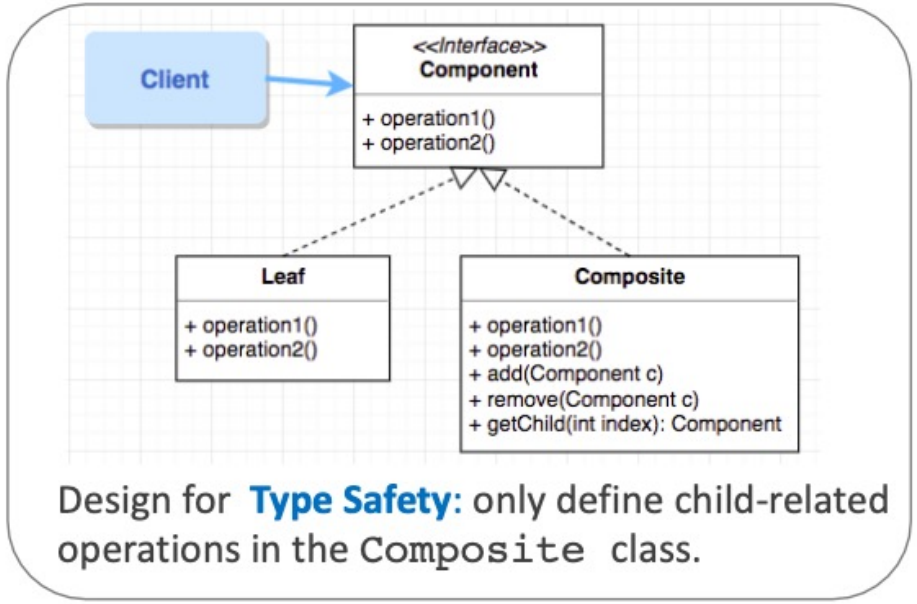
Code Segment from the **`Composite`** class

```
ArrayList<Component>  children = new ArrayList<Component>();

@Override
public double calculateCost() {
    double answer = this.getCost();
    for(Component c : children) {
        answer += c.calculateCost();
    }

    return answer;
}
```

For more, read the example code provided for this week

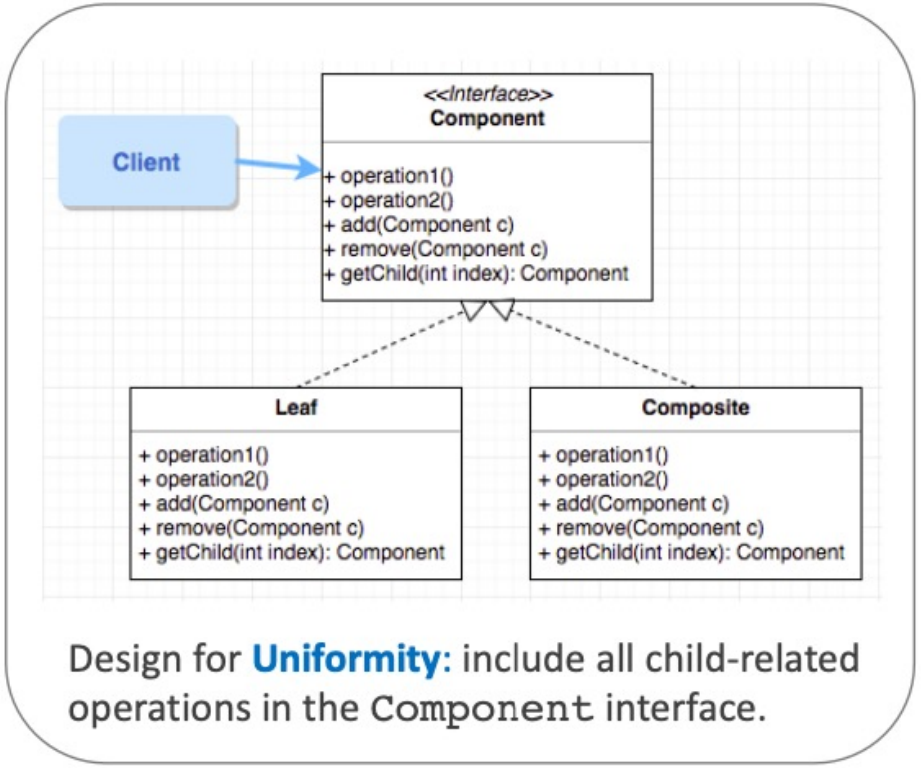
Implementation Issue: Uniformity vs Type Safety

Two possible approaches to implement child-related operations (methods like `add`, `remove`, `getChild`, etc.):



Design for **Type Safety**: only define child-related operations in the `Composite` class.

See the next slide for more details.



Design for **Uniformity**: include all child-related operations in the `Component` interface.

Implementation Issue: Uniformity vs Type Safety

Design for **Uniformity**

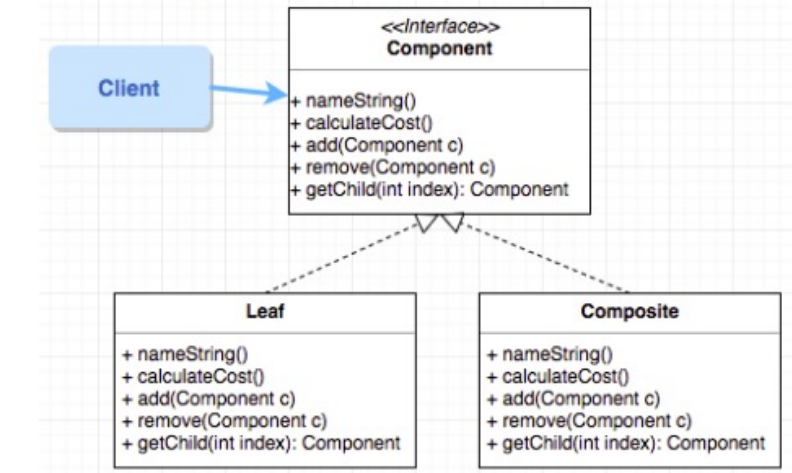
- include all child-related operations in the `Component` interface, this means the `Leaf` class needs to implement these methods with "do nothing" or "throw exception".
- a client can treat both `Leaf` and `Composite` objects uniformly.
- we loose type safety because `Leaf` and `Composite` types are not cleanly separated.
- useful for dynamic structures where children types change dynamically (from `Leaf` to Composite and vice versa), and a client needs to perform child-related operations regularly. For example, a document editor application.

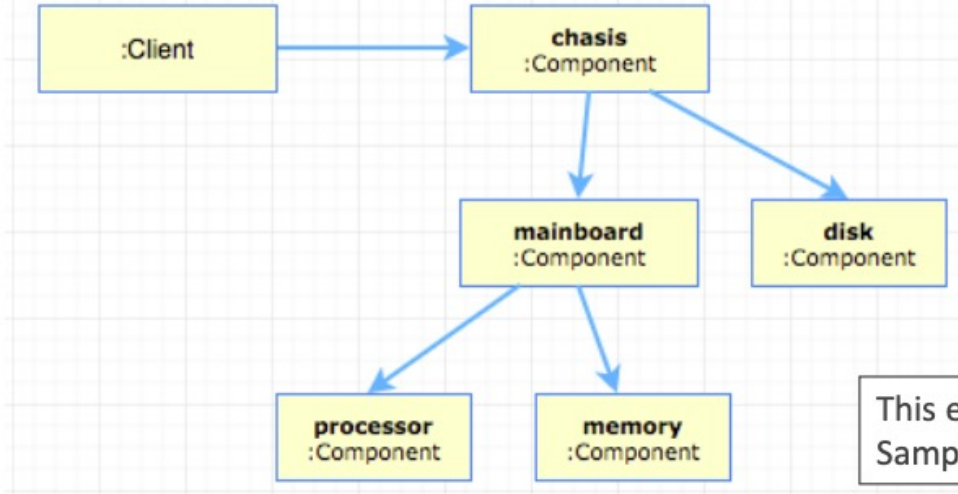
Design for **Type Safety**

- only define child-related operations in the `Composite` class
- the type system enforces type constraints, so a client cannot perform child-related operations on a `Leaf` object.
- a client needs to treat `Leaf` and `Composite` objects differently.
- useful for static structures where a client doesn't need to perform child-related operations on "unknown" objects of type `Component`.

Composite Pattern: Demo Example

Read the example code discussed/developed in the lectures, and also provided for this week






```
Component mainboard = new Composite("Mainboard", 100);
Component processor = new Leaf("Processor", 450);
Component memory    = new Leaf("Memory", 80);
mainboard.add(processor);
mainboard.add(memory);

Component chasis = new Composite("Chasis", 75);
chasis.add(mainboard);

Component disk = new Leaf("Disk", 50);
chasis.add(disk);

System.out.println("[0] "+ processor.nameString());
System.out.println("[0] "+ processor.calculateCost());

System.out.println("[1] "+ mainboard.nameString());
System.out.println("[1] "+ mainboard.calculateCost());

System.out.println("[2] "+ chasis.nameString());
System.out.println("[2] "+ chasis.calculateCost());
```

This example uses design for Uniformity (see composite.uniformity).
Sample code also includes design for Type Safety (see composite.typesafe).

UNSW SYDNEY

Composite Pattern: Demo Example

Read the example code discussed/developed in the lectures, and also provided for this week

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");
        MenuComponent coffeeMenu = new Menu("COFFEE MENU", "S

        MenuComponent allMenus = new Menu("ALL MENUS", "All m

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        pancakeHouseMenu.add(new MenuItem(
            "K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            true,
            2.99));
        pancakeHouseMenu.add(new MenuItem(
            "Regular Pancake Breakfast",
            "Pancakes with fried eggs, sausage",
            false,
            2.99));
```

```
allMenus.print();
```

```
ALL MENUS, All menus combined
---------------------

PANCAKE HOUSE MENU, Breakfast
---------------------
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Regular Pancake Breakfast, 2.99
     -- Pancakes with fried eggs, sausage
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries

DINER MENU, Lunch
---------------------
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  BLT, 2.99
     -- Bacon with lettuce & tomato on whole wheat
  Soup of the day, 3.29
     -- A bowl of the soup of the day, with a side of potato salad
  Hotdog, 3.05
```

Demos .....

- Live Demos ...
- Make sure you properly understand the demo example code available for this week.

Summary

- The Composite Pattern provides a structure to hold both individual objects and composites.
- The Composite Pattern allows clients to treat composites and individual objects uniformly.
- A Component is any object in a Composite structure. Components may be other composites or leaf nodes.
- There are many design tradeoffs in implementing Composite. You need to balance transparency/uniformity and type safety with your needs.