

The Composite Design Pattern



EECS3311 M: Software Design
Winter 2019

CHEN-WEI WANG

Motivating Problem (1)

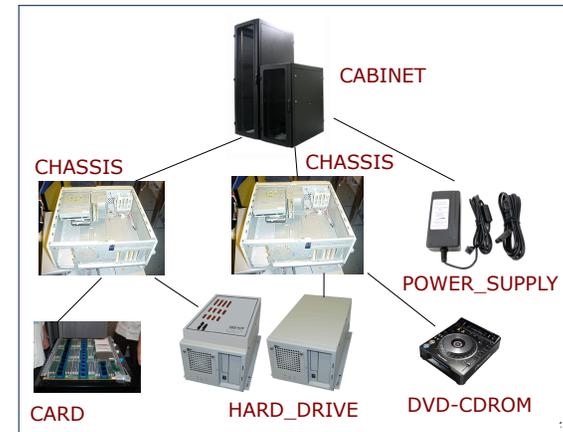


- Many manufactured systems, such as computer systems or stereo systems, are composed of **individual components** and **sub-systems** that contain components.
 - e.g., A computer system is composed of:
 - Individual pieces of equipment (*hard drives, cd-rom drives*)
 - Each equipment has **properties**: e.g., power consumption and cost.
 - Composites such as *cabinets, busses, and chassis*
 - Each *cabinet* contains various types of *chassis*, each of which in turn containing components (*hard-drive, power-supply*) and *busses* that contain *cards*.
- Design a system that will allow us to easily **build** systems and **calculate** their total cost and power consumption.

Motivating Problem (2)



Design for *tree structures* with whole-part *hierarchies*.

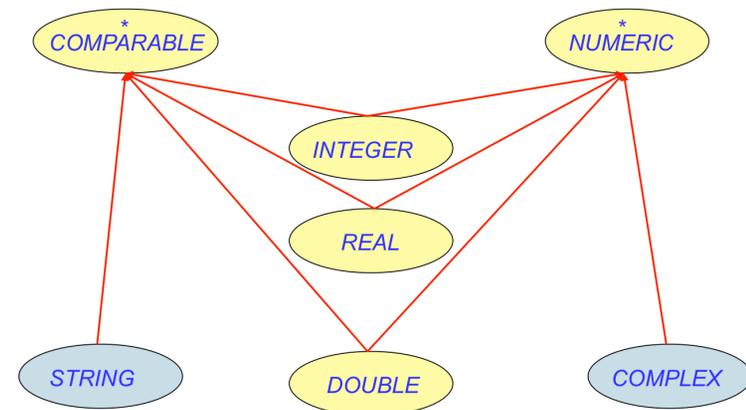


Challenge: There are **base** and **recursive** modelling artifacts.

Multiple Inheritance: Combining Abstractions (1)

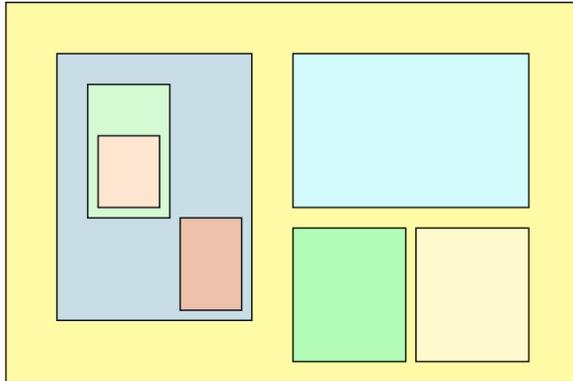


A class may have two more parent classes.



MI: Combining Abstractions (2.1)

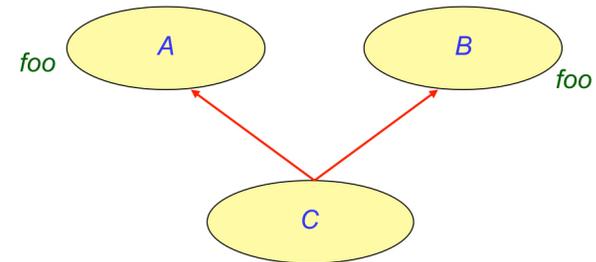
Q: How do you design class(es) for nested windows?



Hints: height, width, xpos, ypos, change width, change height, move, parent window, descendant windows, add child window

5 of 20

MI: Name Clashes



In class C, feature `foo` inherited from ancestor class A clashes with feature `foo` inherited from ancestor class B.

7 of 20

MI: Combining Abstractions (2)

A: Separating *Graphical* features and *Hierarchical* features

```
class RECTANGLE
  feature -- Queries
    width, height: REAL
    xpos, ypos: REAL
  feature -- Commands
    make (w, h: REAL)
    change_width
    change_height
    move
end
```

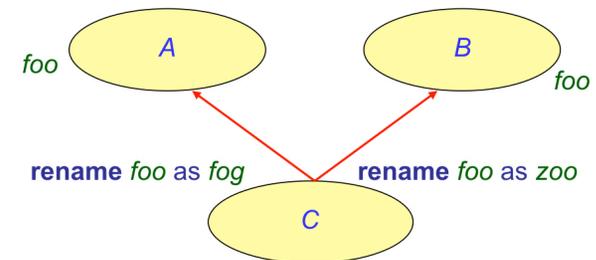
```
class TREE[G]
  feature -- Queries
    parent: TREE[G]
    descendants: LIST[TREE[G]]
  feature -- Commands
    add_child (c: TREE[G])
end
```

```
class WINDOW
  inherit
    RECTANGLE
    TREE[WINDOW]
  feature
    add (w: WINDOW)
end
```

```
test_window: BOOLEAN
local w1, w2, w3, w4: WINDOW
do
  create w1.make(8, 6) ; create w2.make(4, 3)
  create w3.make(1, 1) ; create w4.make(1, 1)
  w2.add(w4) ; w1.add(w2) ; w1.add(w3)
  Result := w1.descendants.count = 2
end
```

6 of 20

MI: Resolving Name Clashes



```
class C
  inherit
    A rename foo as fog end
    B rename foo as zoo end
  ...
```

	o.foo	o.fog	o.zoo
o: A	✓	✗	✗
o: B	✓	✗	✗
o: C	✗	✓	✓

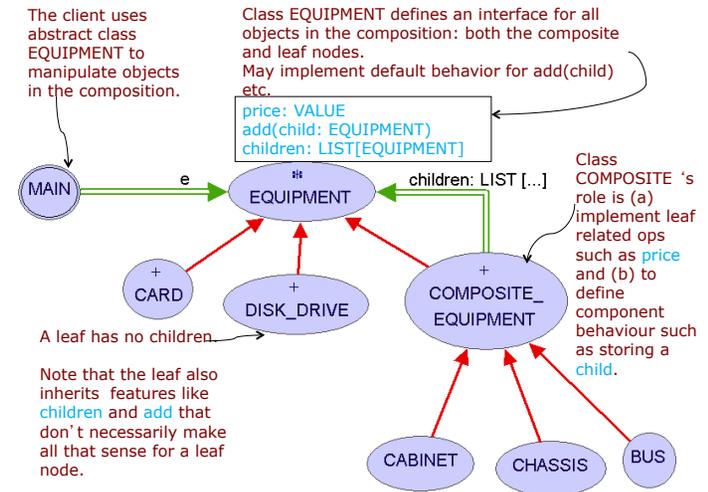
8 of 20

Solution: The Composite Pattern

- Design**: Categorize into *base* artifacts or *recursive* artifacts.
- Programming**:
Build a **tree structure** representing the whole-part **hierarchy**.
- Runtime**:
Allow clients to treat *base* objects (leaves) and *recursive* compositions (nodes) **uniformly**.
⇒ **Polymorphism**: *leaves* and *nodes* are “substitutable”.
⇒ **Dynamic Binding**: Different versions of the same operation is applied on *individual objects* and *composites*.
e.g., Given `e: EQUIPMENT`:
 - `e.price` may return the unit price of a *DISK_DRIVE*.
 - `e.price` may sum prices of a *CHASSIS*’ containing equipments.

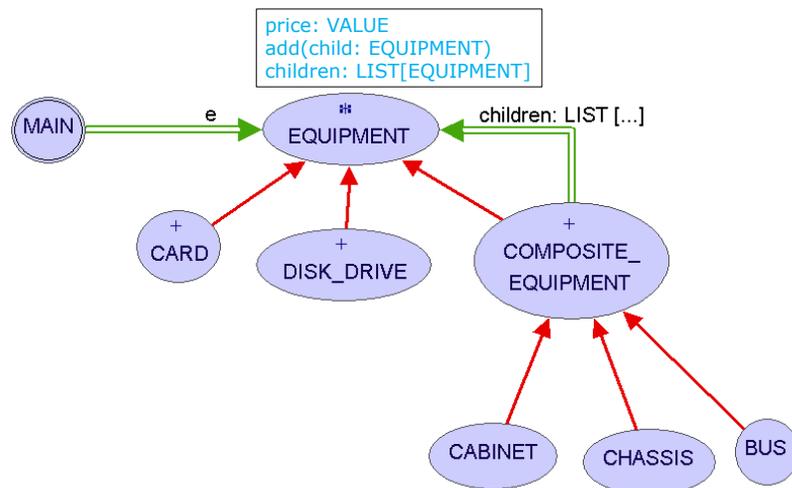
9 of 20

Composite Architecture: Design (1.2)



11 of 20

Composite Architecture: Design (1.1)



10 of 20

Composite Architecture: Design (1.3)

Q: Any flaw of this first design?

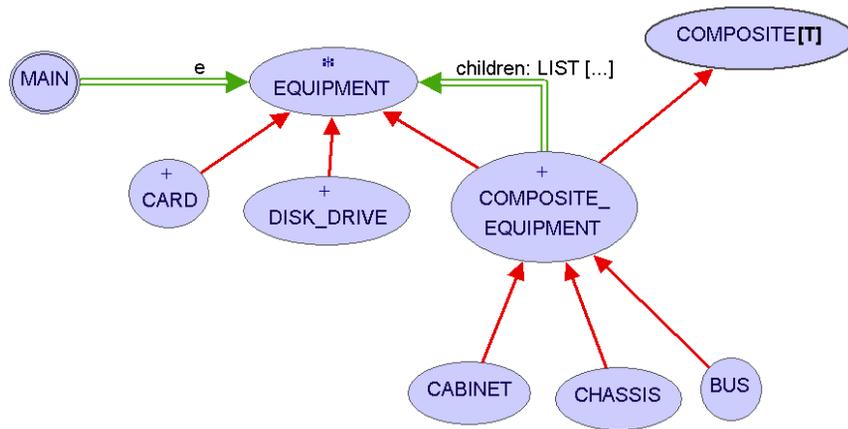
A: Two “composite” features defined at the EQUIPMENT level:

- `children: LIST[EQUIPMENT]`
- `add(child: EQUIPMENT)`

⇒ Inherited to all *base* equipments (e.g., HARD_DRIVE) that do not apply to such features.

12 of 20

Composite Architecture: Design (2.1)



13 of 20

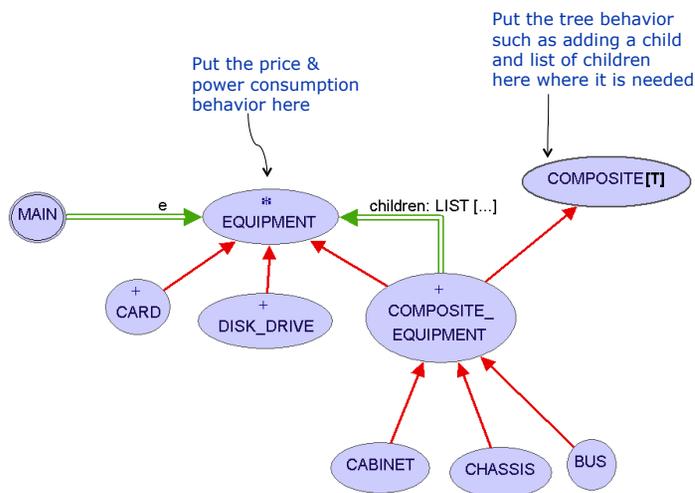
Implementing the Composite Pattern (1)

```
deferred class
    EQUIPMENT
feature
    name: STRING
    price: REAL -- uniform access principle
end
```

```
class
    CARD
inherit
    EQUIPMENT
feature
    make (n: STRING; p: REAL)
    do
        name := n
        price := p -- price is an attribute
    end
end
```

15 of 20

Composite Architecture: Design (2.2)



14 of 20

Implementing the Composite Pattern (2.1)

```
deferred class
    COMPOSITE[T]
feature
    children: LINKED_LIST[T]

    add (c: T)
    do
        children.extend (c) -- Polymorphism
    end
end
```

Exercise: Make the COMPOSITE class *iterable*.

16 of 20

Implementing the Composite Pattern (2.2)



```
class
  COMPOSITE_EQUIPMENT
inherit
  EQUIPMENT
  COMPOSITE [EQUIPMENT]
create
  make
feature
  make (n: STRING)
  do name := n ; create children.make end
  price : REAL -- price is a query
  -- Sum the net prices of all sub-equipments
  do
    across
      children as cursor
    loop
      Result := Result + cursor.item.price -- dynamic binding
    end
  end
end
```

17 of 20

Testing the Composite Pattern



```
test_composite_equipment: BOOLEAN
local
  card, drive: EQUIPMENT
  cabinet: CABINET -- holds a CHASSIS
  chassis: CHASSIS -- contains a BUS and a DISK_DRIVE
  bus: BUS -- holds a CARD
do
  create {CARD} card.make("16Mbs Token Ring", 200)
  create {DISK_DRIVE} drive.make("500 GB harddrive", 500)
  create bus.make("MCA Bus")
  create chassis.make("PC Chassis")
  create cabinet.make("PC Cabinet")

  bus.add(card)
  chassis.add(bus)
  chassis.add(drive)
  cabinet.add(chassis)
  Result := cabinet.price = 700
end
```

18 of 20

Index (1)



- Motivating Problem (1)
- Motivating Problem (2)
- Multiple Inheritance:
- Combining Abstractions (1)
- MI: Combining Abstractions (2.1)
- MI: Combining Abstractions (2)
- MI: Name Clashes
- MI: Resolving Name Clashes
- Solution: The Composite Pattern
- Composite Architecture: Design (1.1)
- Composite Architecture: Design (1.2)
- Composite Architecture: Design (1.3)
- Composite Architecture: Design (2.1)
- Composite Architecture: Design (2.2)

19 of 20

Index (2)



- Implementing the Composite Pattern (1)
- Implementing the Composite Pattern (2.1)
- Implementing the Composite Pattern (2.2)
- Testing the Composite Pattern

20 of 20