Composite Pattern

Design Patterns

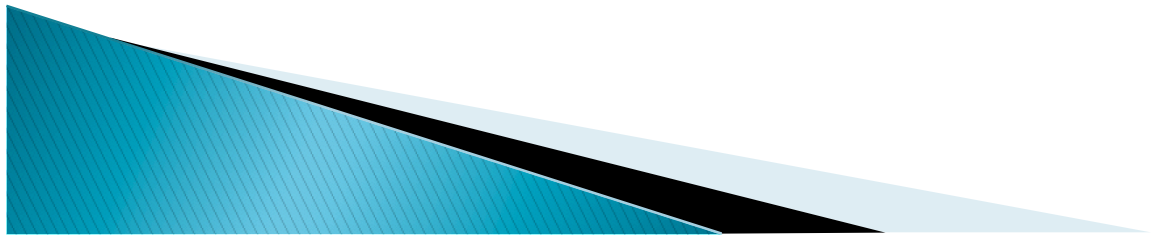
Vasileios Theodorou

Composite Pattern

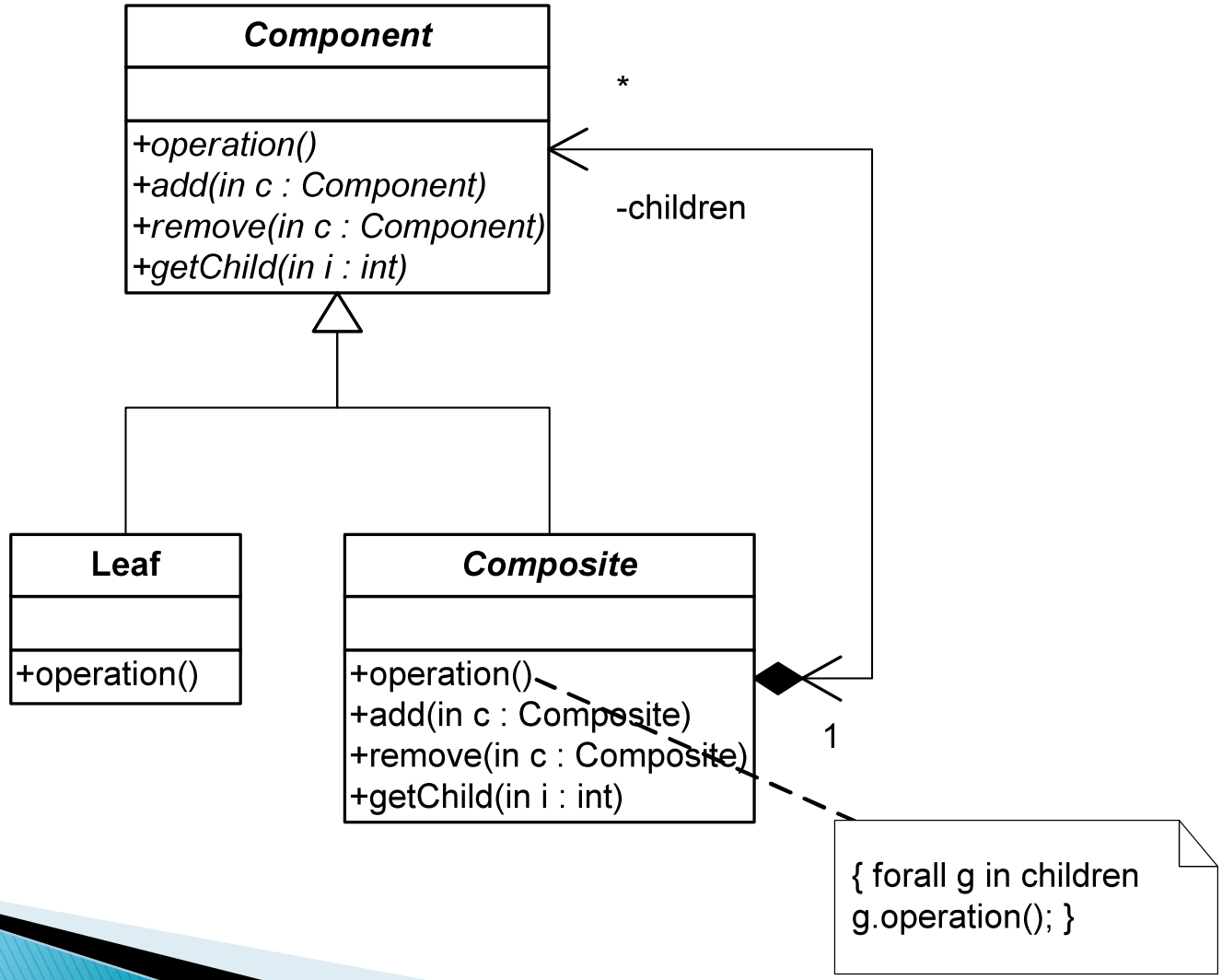
- A partitioning structural design pattern

- **Motivation**
  - Treat individual objects and compositions of objects uniformly

- **Application**
  - Objects are recursively composed
  - Composite objects are treated as a collection of particular, discrete objects
  - Objects are composed into tree structures that represent whole-part hierarchies
  - No discrimination between manipulating simple or composite objects (composite objects can take part in compositions with other objects)
  - Multiple different compositions of Objects: Components may keep track of multiple parents – determine proper parent every time according to context

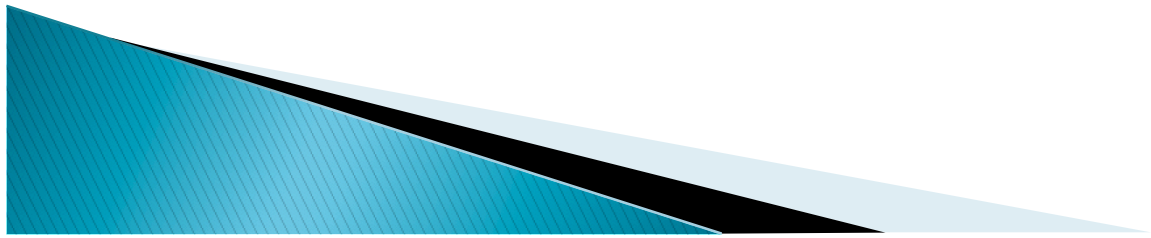


Composite Pattern–Class Diagram

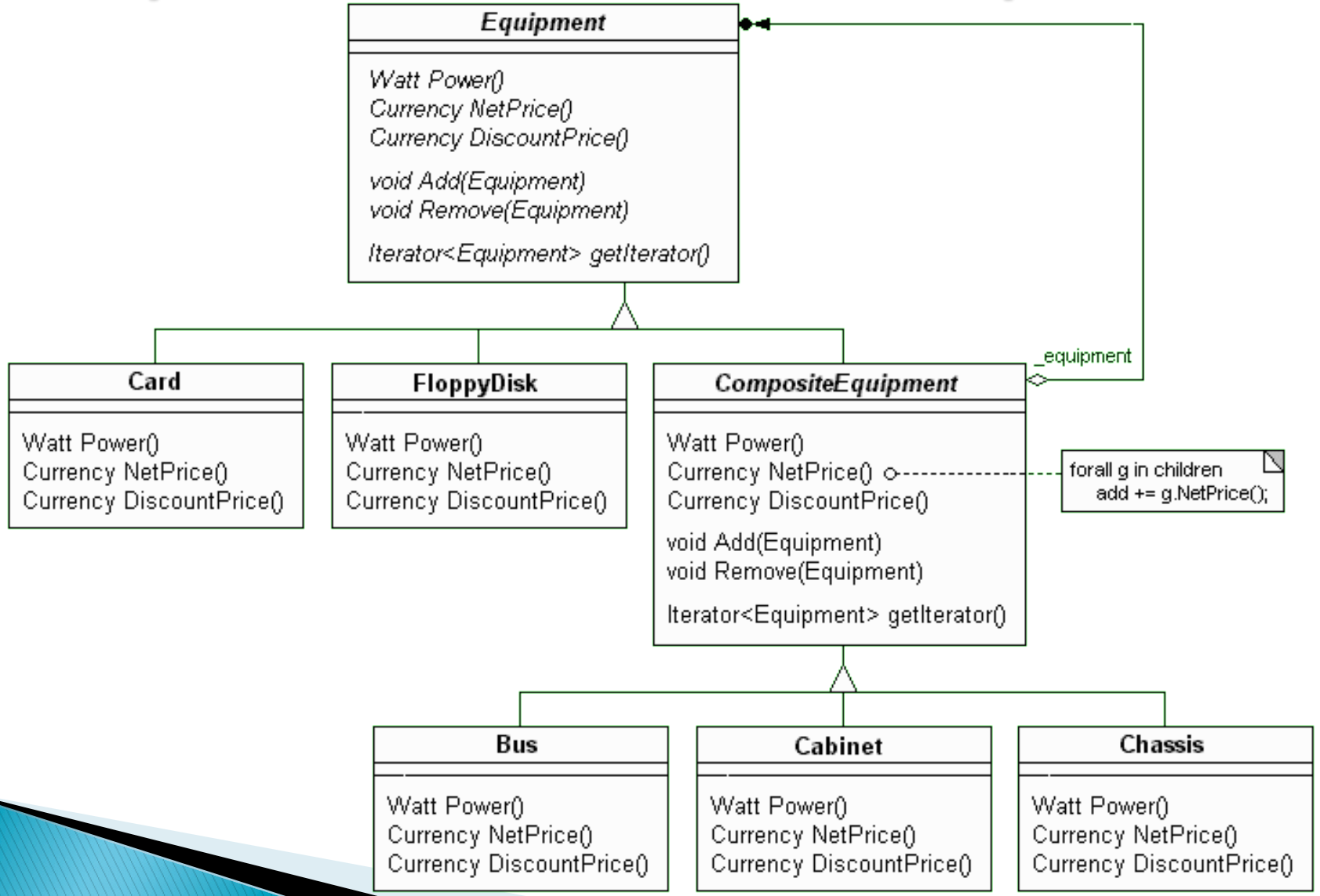


Composite Pattern-Use

- There is a tree/nested structure where objects have at least one common explicit or implicit behavior (e.g. a common method)
- This behavior can be elicited as a part-whole relationship based on the structure
- Objects can dynamically be added to/removed from the structure
- Multi-level analysis on complex Objects
- Specific Uses
  - File System implementation
  - Graphics Editors
  - Dynamic on-line application sessions (build your own pc/car/…)
  - …



Composite Pattern-Example



Composite Pattern–Example Code (1)

```
import java.util.List;
import java.util.ArrayList;

/** "Component" */
interface Equipment {
    public Watt power();
    public Currency netPrice();
    public Currency discountPrice();
}

/** "Composite" */
class CompositeEquipment implements
Equipment {
    private Currency netPrice;
    //Collection of child equipments.
    private List<Equipment> mChildEquipments =
new ArrayList<Equipment>();
    public Currency netPrice() {
      Currency netPrice = this.netPrice;
        for (Equipment equipment :
mChildEquipments) {
            netPrice += equipment.netPrice();
        }
      return(netPrice);
    }
    public Watt power() {
      …
    }
    public Currency discountPrice() {
      …
    }
```

```
 //Adds the equipment to the composition.
   public void add(Equipment equipment) {
      mChildEquipments.add(equipment);
   }
     //Removes the equipment from the
composition.
     public void remove(Equipment equipment) {
         mChildEquipments.remove(equipment);
     }
}

/** "Composite" Components */
class Bus extends CompositeEquipment {
        …
     }
class Cabinet extends CompositeEquipment {
        …
     }
class Chassis extends CompositeEquipment {
        …
     }
```

Composite Pattern–Example Code (2)

```
/** "Leaf" Components */
class Card implements Equipment {
    private Currency netPrice;
    public Currency netPrice() {
      return(this.netPrice);}
    public Watt power() {
      …
    }
    public Currency discountPrice() {
      …
    }
}

class FloppyDisk implements Equipment {
   private Currency netPrice;
   public Currency netPrice() {
     return(this.netPrice);}
   public Watt power() {
     …
   }
   public Currency discountPrice() {
     …
   }
}
```

```
 /** Client Code */
public class Program {

   public static void main(String[] args) {
      Cabinet pcCabinet = new Cabinet();
      Chassis pcChassis = new Chassis();
      pcCabinet.add(pcChassis);
      Bus mcaBus = new Bus();
      Card card = new Card();
      mcaBus.add(card);
      pcChassis.add(mcaBus);
      FloppyDisc floppyD = new FloppyDisc();
      pcChassis.add(floppyD);

System.out.println("%f" , pcChassis.netPrice.value);
   }
}
```

Composite Pattern–Pros & Cons

- **Consequences**

+ Uniformity: All objects in a composite construction are treated the same way
+ Scalability: We can (dynamically, during run-time) create new compositions from existing objects
+ We can analyse a construction at whichever level desired, each component can be seen as a construction itself

- Operating Cost: We might require many other objects for the creation of a composite object

- Inappropriate when:
  - Memory unavailable or performance is the key objective
  - Objects are heterogeneous or non-hierarchical
- Alternative approaches: structure represented virtually, collection of objects

Composite Pattern–Implementation Comments

- Do components know their parents?
- Can we add components which are already part of the structure?
- Same interface for Simple and Composite objects?
- How can an object be identified as a leaf or composite? Use a Boolean? Hard-coded?
- Actual object or reference for the representation of each object composing the Composite object?
- Which object is responsible for the deletion of composing objects? At which point?

Compared to Other Patterns

- **Decorator**
  - Different purpose : dynamically add functionality & clients of class don't need to know X build a hierarchical structure so that the entire structure is viewed as a single entity
  - Similar implementation : Object embedded within another but one instance of a Decorator can only "decorate" one object at a time, not a collection
  - Decorators can be used to decorate different components that all share the same interface
- **Chain of responsibility**
  - Both can call each of their contained objects in turn
  - Chain of responsibility : No discrimination between complex and individual objects
- **Visitor & Flyweight**
  - Collections of objects
  - Passed-in objects influence the overall behavior
  - Visitors can be used to traverse different objects of the Composite Structure
- **Iterator**
  - Usually use one or many of the collection properties of this pattern

Composite Pattern–Questions?