Composite Pattern

Motivation - Composite Pattern

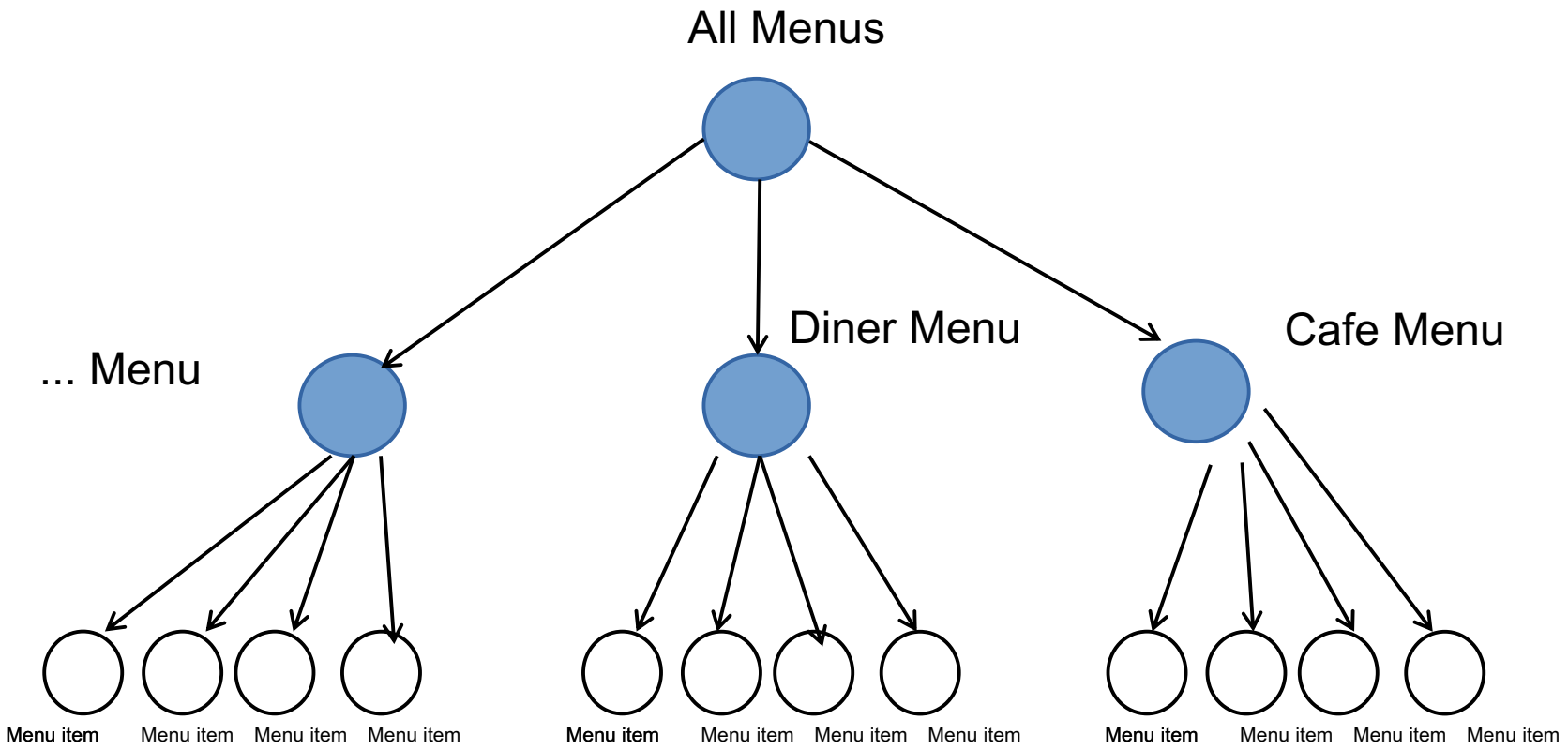
You have a group of objects that can be organized in **tree structure as part-whole hierarchies**.

You want to encapsulate the complexity of traversing through objects in a tree structure.

You want to **handle leaves and nodes equally**

Example – Restaurant Menu

Which vegetarian dishes a restaurant has?



Use Case – Waitress

```
public class Waitress {
        MenuComponent allMenus;

        public Waitress(MenuComponent allMenus) {  this.allMenus = allMenus; }
        public void printMenu() { allMenus.print();  }

        public void printVegetarianMenu() {
                Iterator<MenuComponent> iterator = allMenus.createIterator();

                // Print all vegan menu items
                while (iterator.hasNext()) {
                        MenuComponent menuComponent = iterator.next();

                        try {
                                if (menuComponent.isVegetarian()) {
                                        menuComponent.print();
                                }
                        } catch (  UnsupportedOperationException e  ) {
                        e.stacktrace();
                        }
                }
        } }
```

Use Case 1 – Menu and MenuComponent

```
public abstract class MenuComponent {
        public void add(MenuComponent menuComponent) {
                throw new UnsupportedOperationException();
        }
        public void remove(MenuComponent menuComponent) {
                throw new UnsupportedOperationException();
        }
        public MenuComponent getChild(int i) {
                throw new UnsupportedOperationException();
        }
        public String getName() {
                throw new UnsupportedOperationException();
        }
        public String getDescription() {
                throw new UnsupportedOperationException();
        }
        public double getPrice() {
                throw new UnsupportedOperationException();
        }
        public boolean isVegetarian() {
                throw new UnsupportedOperationException();
        }
        public void print() {
                throw new UnsupportedOperationException();
        }
}
```

Because some of these methods only make sense for MenuItems, and some others Menus, the default implementation is **UnsupportedOperationException**

Use Case 1 – Menu and MenuComponent

```
public class Menu extends MenuComponent {

        List<MenuComponent> menuComponents = new ArrayList<MenuComponent>();

        String name;
        String description;
    ….
}
```

(Details of Class Menu on the next slide)

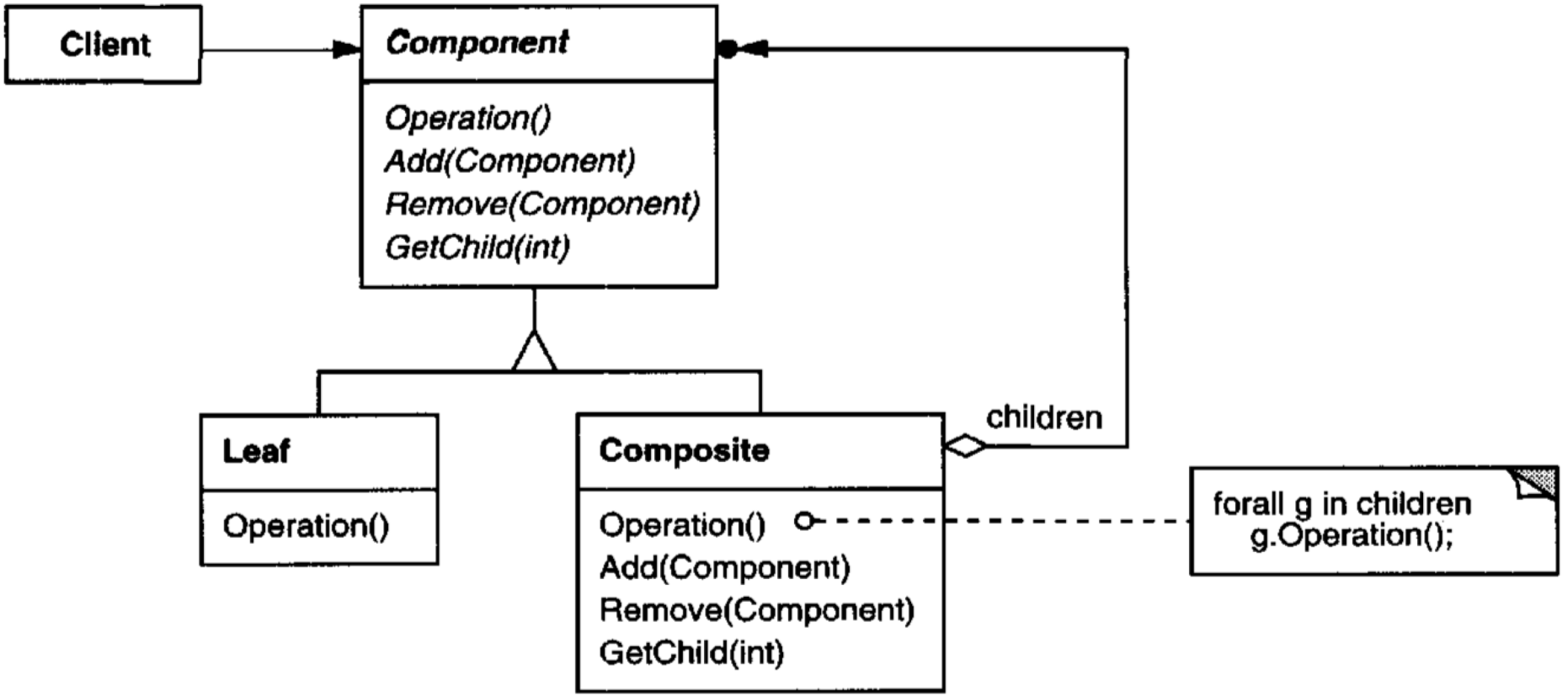
```
public class MenuItem extends MenuComponent {
        String name;
        String description;
        boolean vegetarian;
        double price;

        public MenuItem( … ) {
        }
        public String getName() { return name; }
        public String getDescription() { return description; }
        public double getPrice() { return price; }
        public boolean isVegetarian() { return vegetarian;}
        public void print() { … }
}
```

Use Case 1 – Menu and MenuComponent

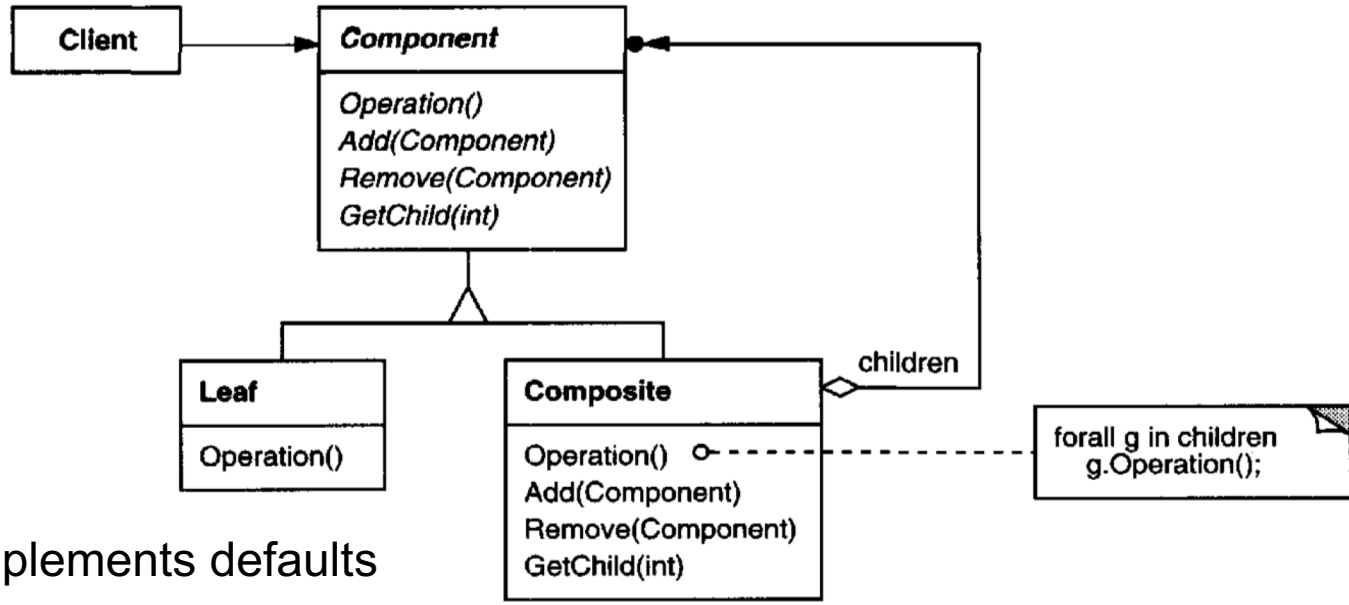
```
public class Menu extends MenuComponent {
  List<MenuComponent> menuComponents = new ArrayList<MenuComponent>();

  String name;
  String description;
  public Menu(String name, String description) { this.name = name; this.description = description;  }
  public void add(MenuComponent menuComponent) {     menuComponents.add(menuComponent);      }
  public void remove(MenuComponent menuComponent) {   menuComponents.remove(menuComponent);      }
  public MenuComponent getChild(int i) {    return (MenuComponent)menuComponents.get(i);    }
…

  public void print() {
            System.out.print("\n" + getName());      System.out.println(", " + getDescription());
            Iterator<MenuComponent> iterator = menuComponents.iterator();
            // HERE IS THE IMPORTANT PART
            while (iterator.hasNext()) {
            MenuComponent menuComponent = (MenuComponent)   iterator.next();
                              menuComponent.print();
            }
  }
}
```

Composite Pattern Definition

Definition: "**Compose** objects into tree structures to represent part-whole hierarchies. **Composite** lets clients treat individual objects and compositions of objects uniformly."



Structure of Composite Pattern



Component:

- declares the interface and implements defaults

- declares an interface for accessing and managing its child components.

Leaf:

- it represents leaf objects in the composition, has no children and defines behavior for primitive objects

Composite:

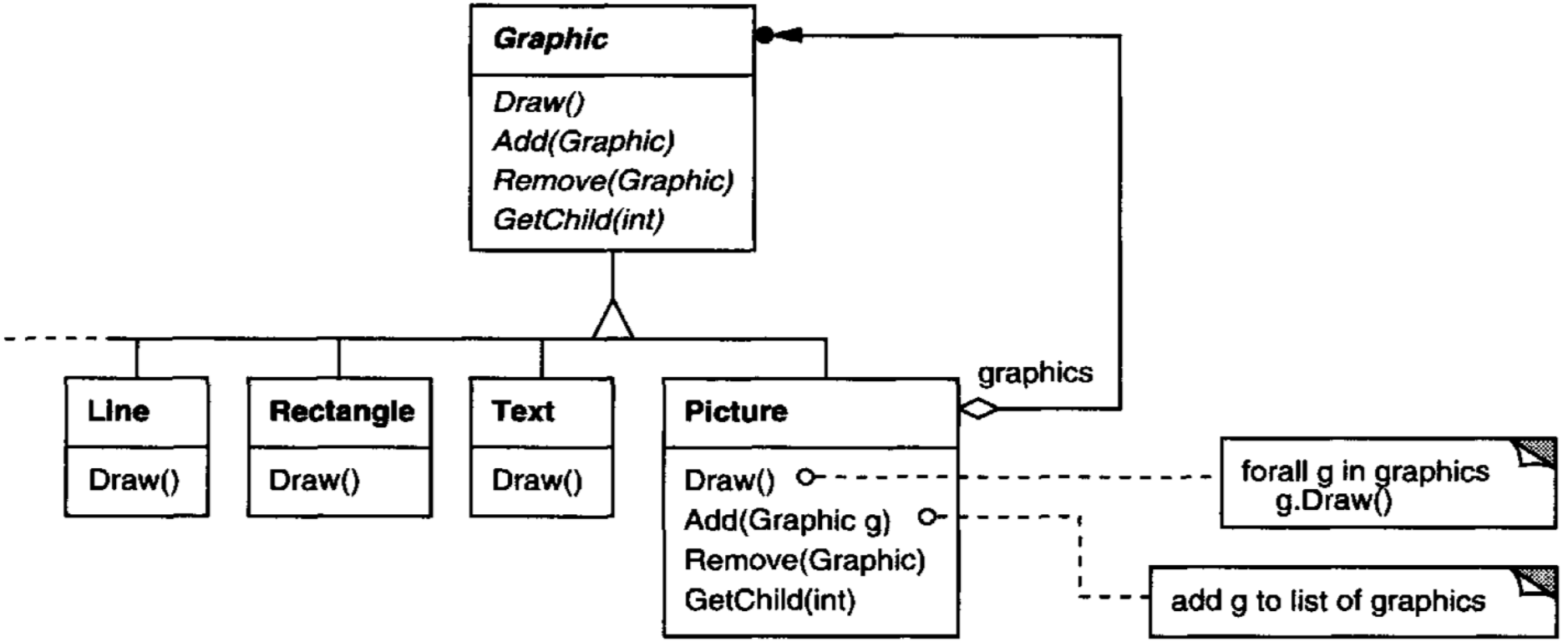
- defines behavior for components having children and stores child components.

- implements child-related operations in the Component interface.

Client:

- manipulates objects in the composition through the Component interface

Another Example



When use the Composite Pattern

You want to **represent part-whole hierarchies** of objects.

You want clients to be able **to ignore the difference between compositions of objects and individual objects**.

Clients will treat all objects in the composite structure **uniformly**.

Consequences of using Composite Pattern

It defines class hierarchies consisting of primitive objects and composite objects.

It makes the client simple. Clients can treat composite structures and individual objects uniformly.

It makes it easier to add new kinds of components.

It can make your design overly general. It is harder to restrict the components of a composite. For example when you want a composite to have only certain components.

Summary of Composite Pattern

"**Compose** objects into tree structures to represent part-whole hierarchies. **Composite** lets clients treat individual objects and compositions of objects uniformly."

Composite Pattern is of type **Structural Patterns**

Related Patterns:

- **Decorator:** When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.
- **Iterator**: can be used to traverse composites.
- **Flyweight**: lets you share components, but they can no longer refer to their parents
- **Visitor:** localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.