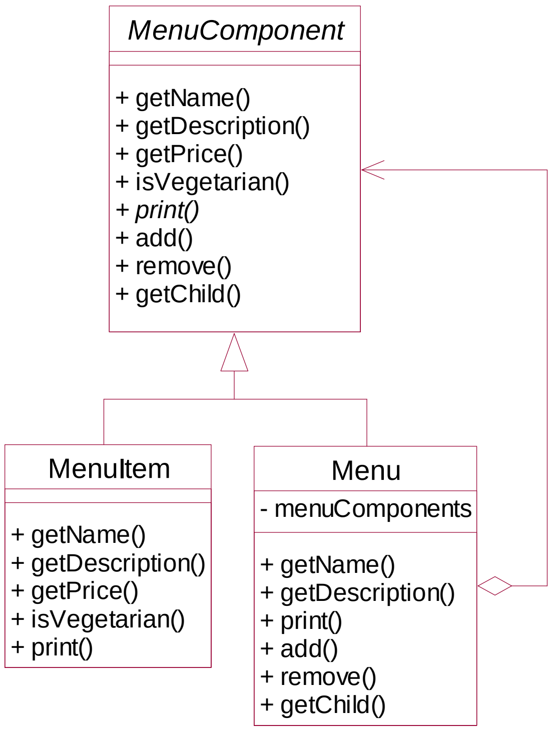
COMPOSITE PATTERN

Problem 1:

1. In previous pattern, Iterator pattern, we discussed about restaurant menu problem. When our restaurant has so many menus and they are organized in a tree-like structure, we will difficult to handle this design. With composite pattern, we'll easy to manage our menus. Using composite pattern means you can compose your object into tree structures to represent part-whole hierarchies.



2. Create **RestaurantMenuWithCP1** project, you have to deal with these problem:
   + Create **`MenuComponent`** abstract class and provide some default implementation for these methods: **`getName()`, `getDescription()`, `getPrice()`, `isVegetarian()`** and **`print()`** method, and **`add()`, `remove()` and `getChild()`** method use to add, remove or get a component. Because some these methods make sense for MenuItem, and some for Menu so the default implementation is **`UnsupportedOperationException`**.

```
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) {
```

```java
        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();
    }
}
```

+ **MenuItem** that extend **MenuComponent** have to override getter methods **getName(), getDescription(), getPrice(), isVegetarian()** and **print()** method used to print the complete entry.

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

    public MenuItem(String name, String description, boolean vegetarian,
            double price) {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }
    // your code here
}
```

+ **Menu** is also a **MenuComponent** too. It has three attributes, an **ArrayList** of **MenuComponent**, a **name** and a **description**. Menu must override these methods.

- **add(), remove(), getChild()**
- **getName(), getDescription()** used to get the name and description of component.
- **print()** used to print the name, description and information of all **Menu's** components. You can use **Iterator** to iterate through the **Menu's** component.
- We needn't override **getPrice()** or **isVegetarian()** method. By default, **UnsupportedOperationException** exception'll be throwed.

```
public class Menu extends MenuComponent {
   ArrayList menuComponents = new ArrayList();
   String name;
   String description;

   public Menu(String name, String description) {
      this.name = name;
      this.description = description;
   }

   public void add(MenuComponent menuComponent) {
      // your code here
   }

   public void remove(MenuComponent menuComponent) {
      // your code here
   }

   public MenuComponent getChild(int i) {
      // your code here
   }

   public String getName() {
      return name;
   }

   public String getDescription() {
      return description;
   }

   public void print() {
      // your code here
   }
}
```

+ **`Waitress`** class now just keeps the top level menu component, the **`printMenu()`** method only call **`print()`** method of **`MenuComponent`** to print the entire menu hierarchy.

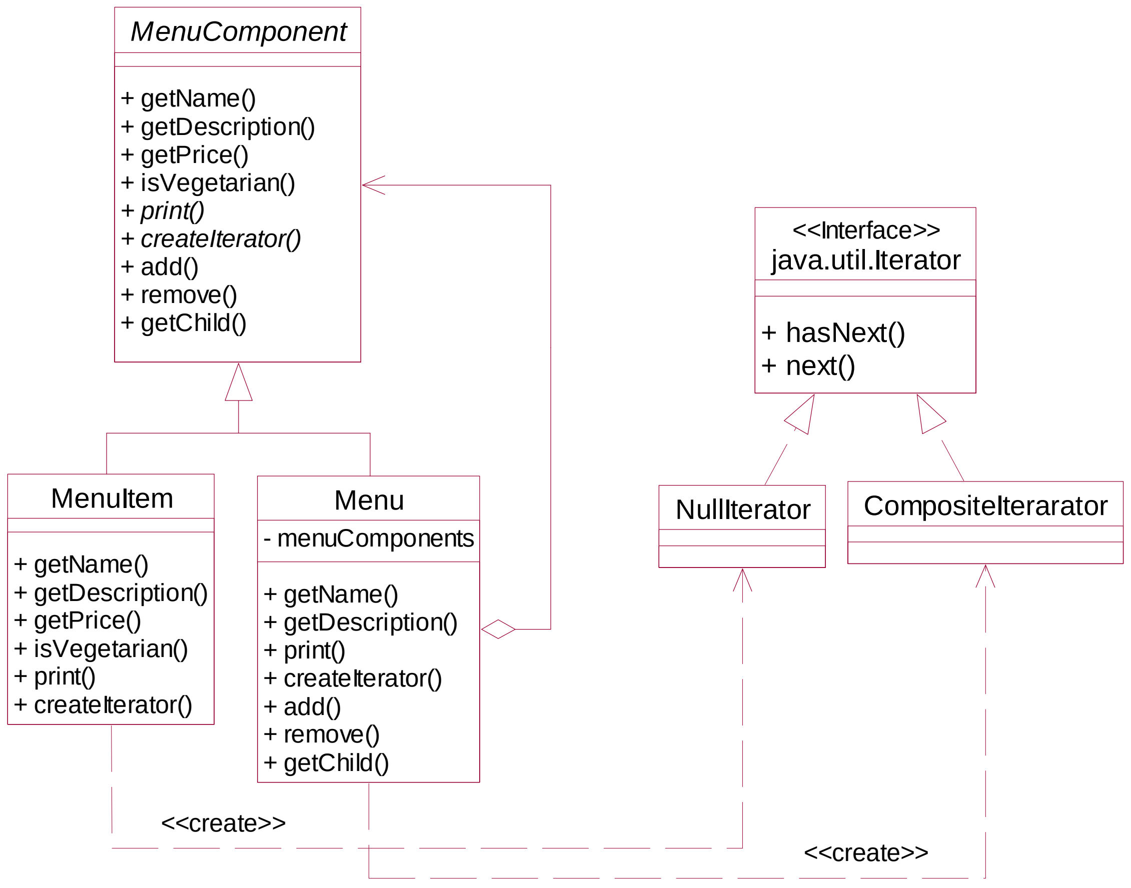
```
public class Waitress {
   MenuComponent allMenus;

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

   public void printMenu() {
      // your code here
   }
}
```

+ Write a test driven to your implementation
  - First of all, you should create all the menu object
  - Create the top level menu, and then you can use **`add()`** method to add all the menu items. The **`Waitress`** hold entire menu hierarchy then she prints out the menu easily.

```java
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",
            "Stuff to go with your afternoon coffee");

      MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

      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));
      pancakeHouseMenu.add(new MenuItem("Blueberry Pancakes",
            "Pancakes made with fresh blueberries, and blueberry syrup", true,
            3.49));
      pancakeHouseMenu.add(new MenuItem("Waffles",
            "Waffles, with your choice of blueberries or strawberries", true,
            3.59));

      dinerMenu.add(new MenuItem("Vegetarian BLT",
            "(Fakin') Bacon with lettuce & tomato on whole wheat", true, 2.99));
      dinerMenu.add(new MenuItem("BLT",
            "Bacon with lettuce & tomato on whole wheat", false, 2.99));
      dinerMenu.add(new MenuItem("Soup of the day",
            "A bowl of the soup of the day, with a side of potato salad",
            false, 3.29));
      dinerMenu.add(new MenuItem("Hotdog",
            "A hot dog, with saurkraut, relish, onions, topped with cheese",
            false, 3.05));
      dinerMenu.add(new MenuItem("Steamed Veggies and Brown Rice",
            "Steamed vegetables over brown rice", true, 3.99));

      dinerMenu.add(new MenuItem("Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true, 3.89));

      dinerMenu.add(dessertMenu);

      dessertMenu.add(new MenuItem("Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla icecream",
            true, 1.59));

      dessertMenu.add(new MenuItem("Cheesecake",
           "Creamy New York cheesecake, with a chocolate graham crust", true,
            1.99));
      dessertMenu.add(new MenuItem("Sorbet",
            "A scoop of raspberry and a scoop of lime", true, 1.89));

      cafeMenu.add(new MenuItem("Veggie Burger and Air Fries",
            "Veggie burger on a whole wheat bun, lettuce, tomato, and fries",
            true, 3.99));
      cafeMenu.add(new MenuItem("Soup of the day",
            "A cup of the soup of the day, with a side salad", false, 3.69));
      cafeMenu.add(new MenuItem("Burrito",
         "A large burrito, with whole pinto beans, salsa, guacamole", true,
            4.29));

      cafeMenu.add(coffeeMenu);

      coffeeMenu.add(new MenuItem("Coffee Cake",
            "Crumbly cake topped with cinnamon and walnuts", true, 1.59));
      coffeeMenu.add(new MenuItem("Bagel",
            "Flavors include sesame, poppyseed, cinnamon raisin, pumpkin",
            false, 0.69));
      coffeeMenu.add(new MenuItem("Biscotti",
            "Three almond or hazelnut biscotti cookies", true, 0.89));

      Waitress waitress = new Waitress(allMenus);

      waitress.printMenu();
   }
}
```

Problem 2:

1. Flashback to the **`Iterator`** – A **`Composite Iterator`**



2. Create **RestaurantMenuWithCP2** which is a copy of **RestaurantMenuWithCP1** then implement these required:
   + Create **`NullIterator`** class, it implements **`java.util.Iterator`**, it implement these method:
     - **`next()`** always returns null. **`hasNext()`** always returns false.
     - **`NullIterator`** wouldn't think of supporting remove() method.

```
public class NullIterator implements Iterator {
   public Object next() {
      // your code here
   }

   public boolean hasNext() {
      // your code here
   }

   public void remove() {
      // your code here
   }
}
```

   + Create **`CompositeIterator`** which implements **`java.util.Iterator`**. **`CompositeIterator's`** attribute is a stack which stores the top level composite. This class has to implement **`next()`**, **`hasNext()`** and **`remove()`** method from **`Iterator`** interface.

- In **hasNext()** method, first of all you would check if the stack is empty or not. If yes, you can return false. Otherwise, you should get the **Iterator** off the top of the stack and see if it has next element. If it doesn't have next element, you can pop it off the stack and call **hasNext()** recursive. Otherwise, it means there is a next element, you can return true.
- When you want to get element by calling **next()** method, you must sure that there is one (you can call **hasNext()** method to check this). If there is a next element, now you get the current iterator and its next element off the stack. If that element is a **Menu**, you must add it into your stack; otherwise you can return that element.
- You shouldn't support **remove()** method, just throw a **UnsupportedOperationException** exception.

```
public class CompositeIterator implements Iterator {
   Stack stack = new Stack();

   public CompositeIterator(Iterator iterator) {
      stack.push(iterator);
   }

   public Object next() {
      if (hasNext()) {
         Iterator iterator = (Iterator) stack.peek();
         MenuComponent component = (MenuComponent) iterator.next();
         if (component instanceof Menu) {
            stack.push(component.createIterator());
         }
         return component;
      } else {
         return null;
      }
   }

   public boolean hasNext() {
      if (stack.empty()) {
         return false;
      } else {
         Iterator iterator = (Iterator) stack.peek();
         if (!iterator.hasNext()) {
            stack.pop();
            return hasNext();
         } else {
            return true;
         }
      }
   }

   public void remove() {
      throw new UnsupportedOperationException();
   }
}
```

+ Add **createIterator()** method into **MenuComponent** abstract class

+ **MenuItem** implementation **createIterator()** method returns **NullIterator**.

- **`Menu`** implementation **`createIterator()`** method returns **`CompositeIterator`**.

- Now, **`Waitress`** class add a new method called **`printVegetarianMenu()`** to print out all vegetarian menu items. First, you must take all menu items, you can use **`createIterator()`** method. Then, iterate through every element and check if it's vegetarian item, you can print it out. Remember, you much catch **`UnsupportedOperationException`** because print() is only called on MenuItems, never composites.

```java
public class Waitress {
   MenuComponent allMenus;

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

   public void printMenu() {
      allMenus.print();
   }

   public void printVegetarianMenu() {
      // your code here
   }
}
```

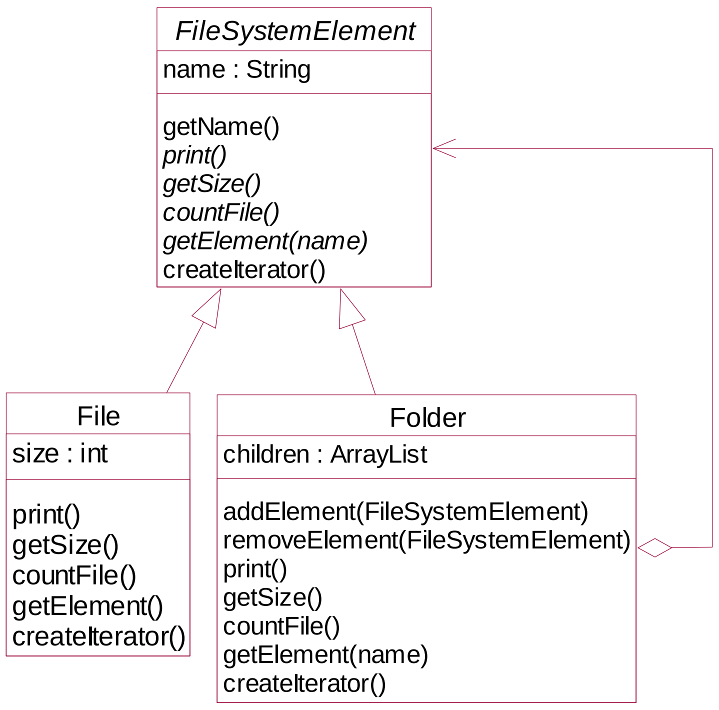
- Test your implementation

Problem 2:

File và folder là các thành phần của một hệ thống quản lý tập tin.

- File biểu diễn dữ liệu cần lưu trữ.
- Folder là ngăn chứa các tập tin. Trong mỗi folder có thể chứa các folder con khác.

Thiết kế cấu trúc File và Folder dùng mẫu composite



Cài đặt các thao tác xử lý sau trên cấu trúc file và folder

1. **`getSize()`**: tính kích thước file / folder
2. **`countFile()`**: Đếm số file trong một folder
3. **`print()`**: In nội dung file / folder
4. **`createIterator()`**: tạo một **`iterator`** để duyệt các file và folder của một **`FileSystemElement`**.

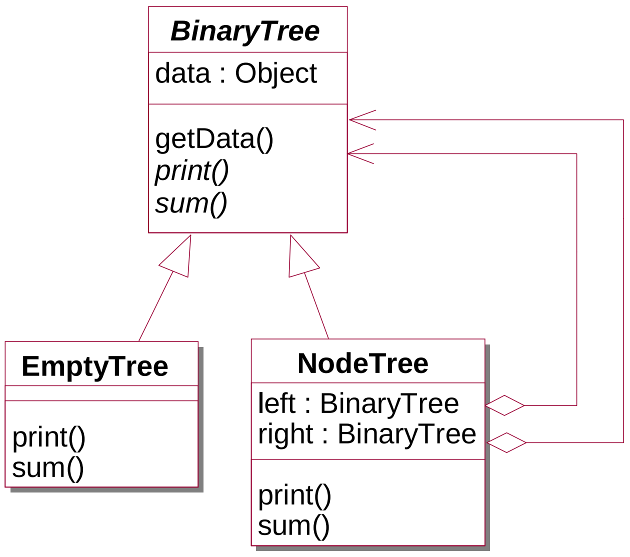
Problem 3:

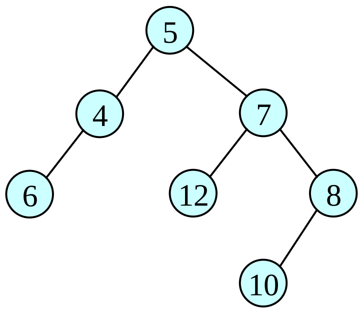
Cấu trúc cây nhị phân (**Binary Tree**)

Một cây nhị phân có thể là:

- **`EmptyTree`**, cây nhị phân không có dữ liệu
- **`NodeTree`**, chứa một đối tượng dữ liệu data, và 2 cây con gọi là left và right.

Chúng ta cài đặt cấu trúc cây nhị phân trên bằng mẫu composite.



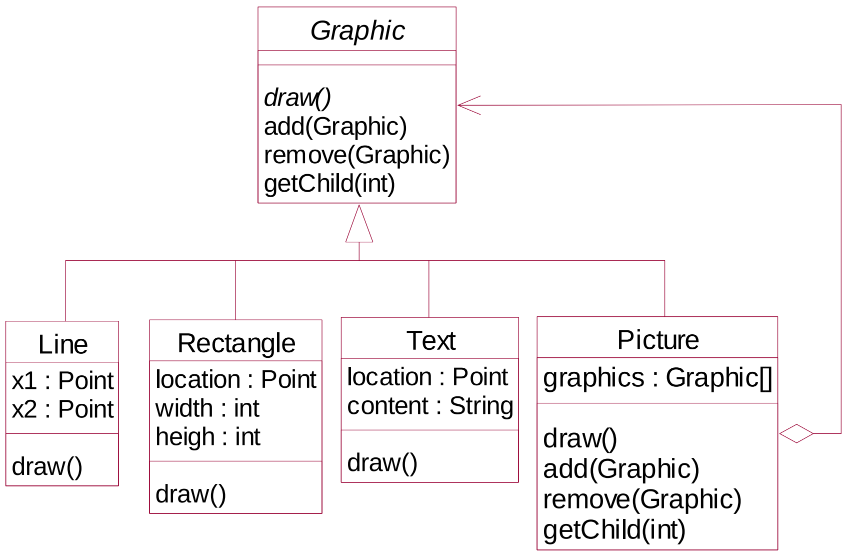


Cài đặt các thao tác xử lý sau trên cấu trúc cây nhị phân

1. **`sum()`**: tính tổng trọng số của cây

2. **`print()`**: In nội dung cây

Problem 4: Graphic hierachy



Cài đặt cấu trúc các thành phần đồ họa như trên và thao tác **`draw()`**