Composite Pattern

Prof. Jonathan Lee（李允中）
Department of CSIE
National Taiwan University

Design Aspect of Composite

Structure and composition of an object

Outline

- Schematic Capture Systems Requirements Statements
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- Recurrent Problems
- Intent
- Composite Pattern Structure
- Another Example: Extended Merge of Two Menus
- BPEL Engine: Another Example
- Homework

Schematic Capture Systems (Composite)

Prof. Jonathan Lee (李允中)

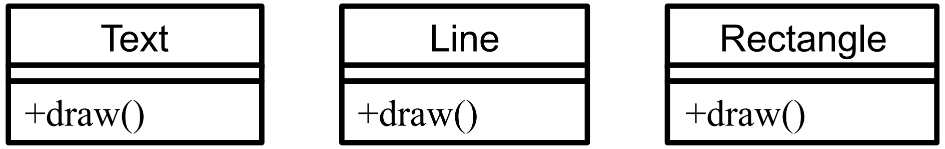
Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statement

- In schematic capture application,
  - There are some basic components can be drawn such as Text, Line, and Rectangle.
  - The user can group basic components to form larger components, which in turn can be grouped to form still larger components.

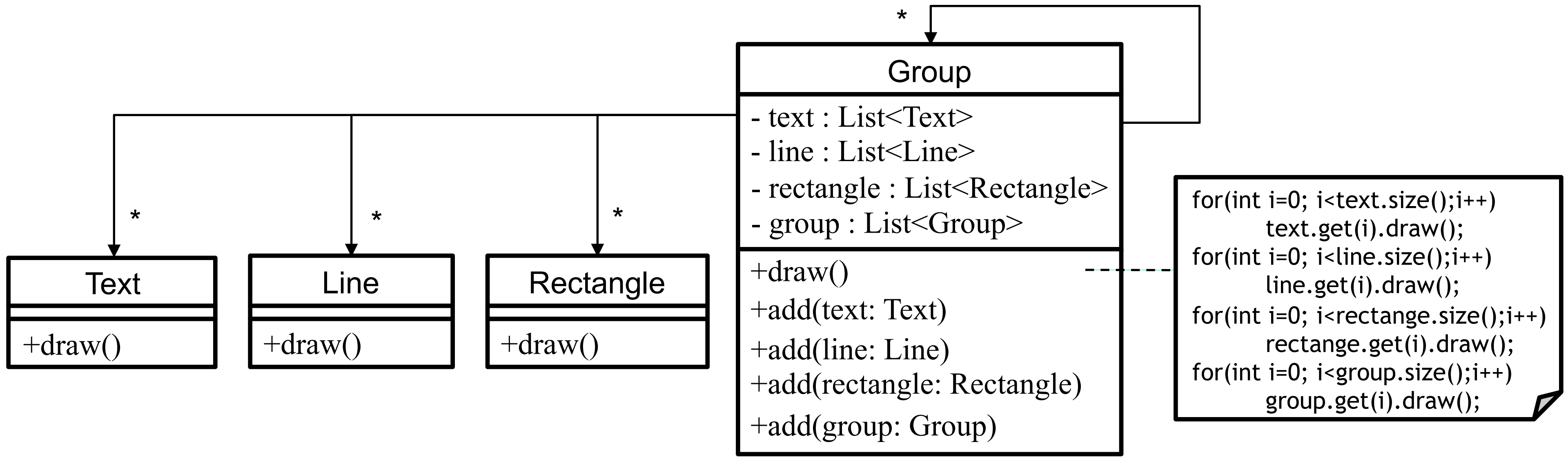
Requirements Statements$_1$

- In schematic capture application, there are some basic components that can be drawn such as Text, Line and Rectangle.



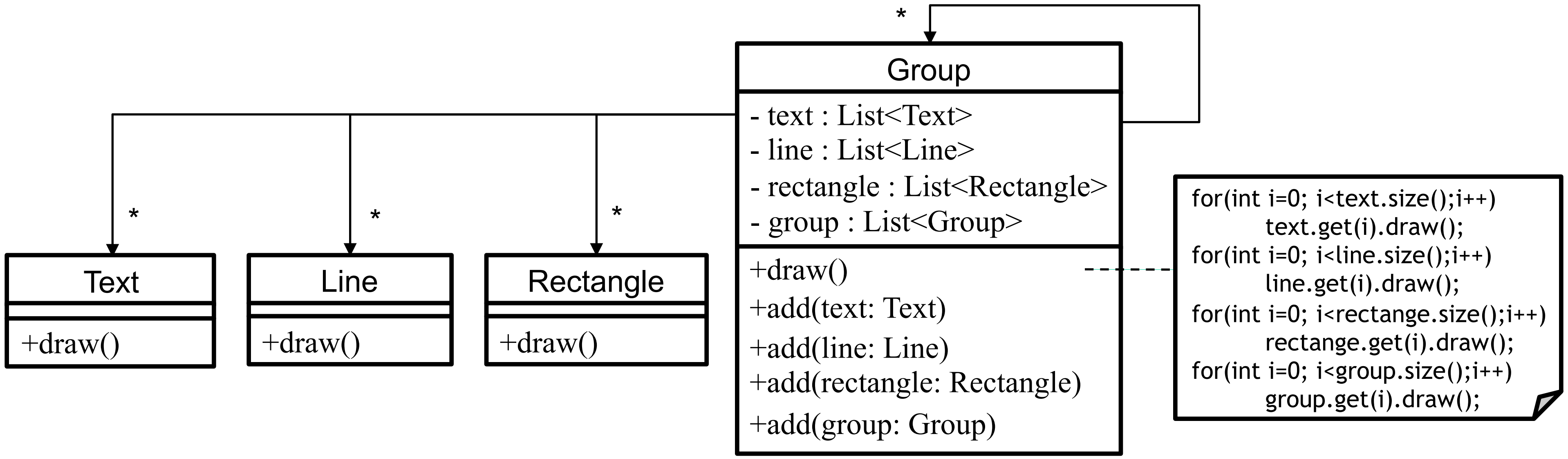
Requirements Statements$_2$

- The user can group basic components to form larger components, which in turn can be grouped to form still larger components.



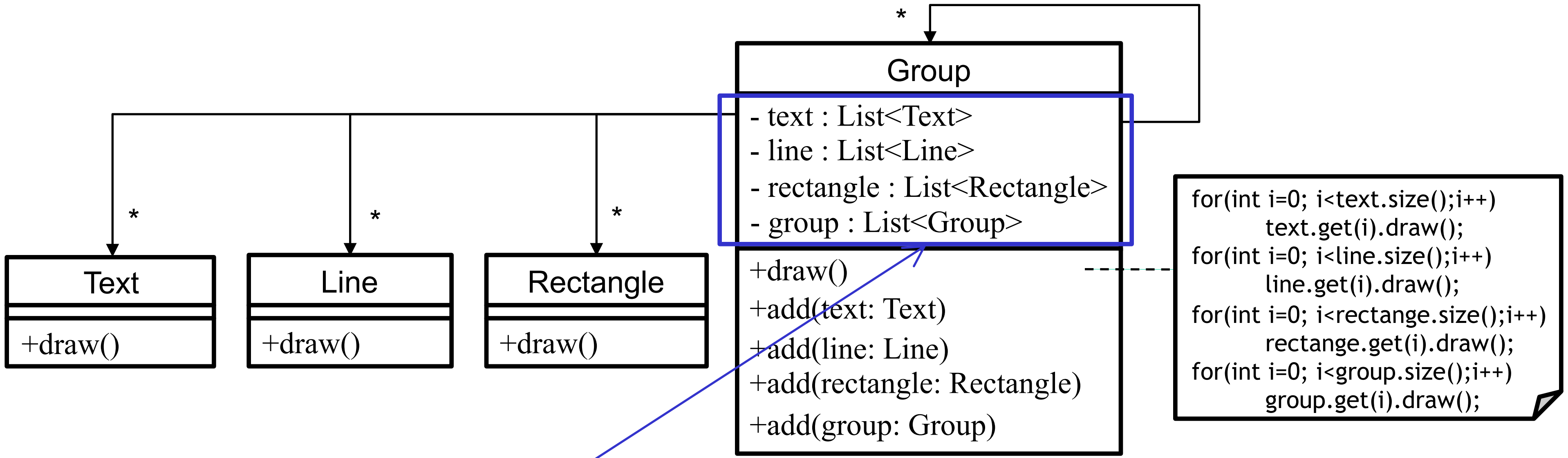
```
for(int i=0; i<text.size();i++)
        text.get(i).draw();
for(int i=0; i<line.size();i++)
        line.get(i).draw();
for(int i=0; i<rectange.size();i++)
        rectange.get(i).draw();
for(int i=0; i<group.size();i++)
        group.get(i).draw();
```

Initial Design



```
for(int i=0; i<text.size();i++)
        text.get(i).draw();
for(int i=0; i<line.size();i++)
        line.get(i).draw();
for(int i=0; i<rectange.size();i++)
        rectange.get(i).draw();
for(int i=0; i<group.size();i++)
        group.get(i).draw();
```

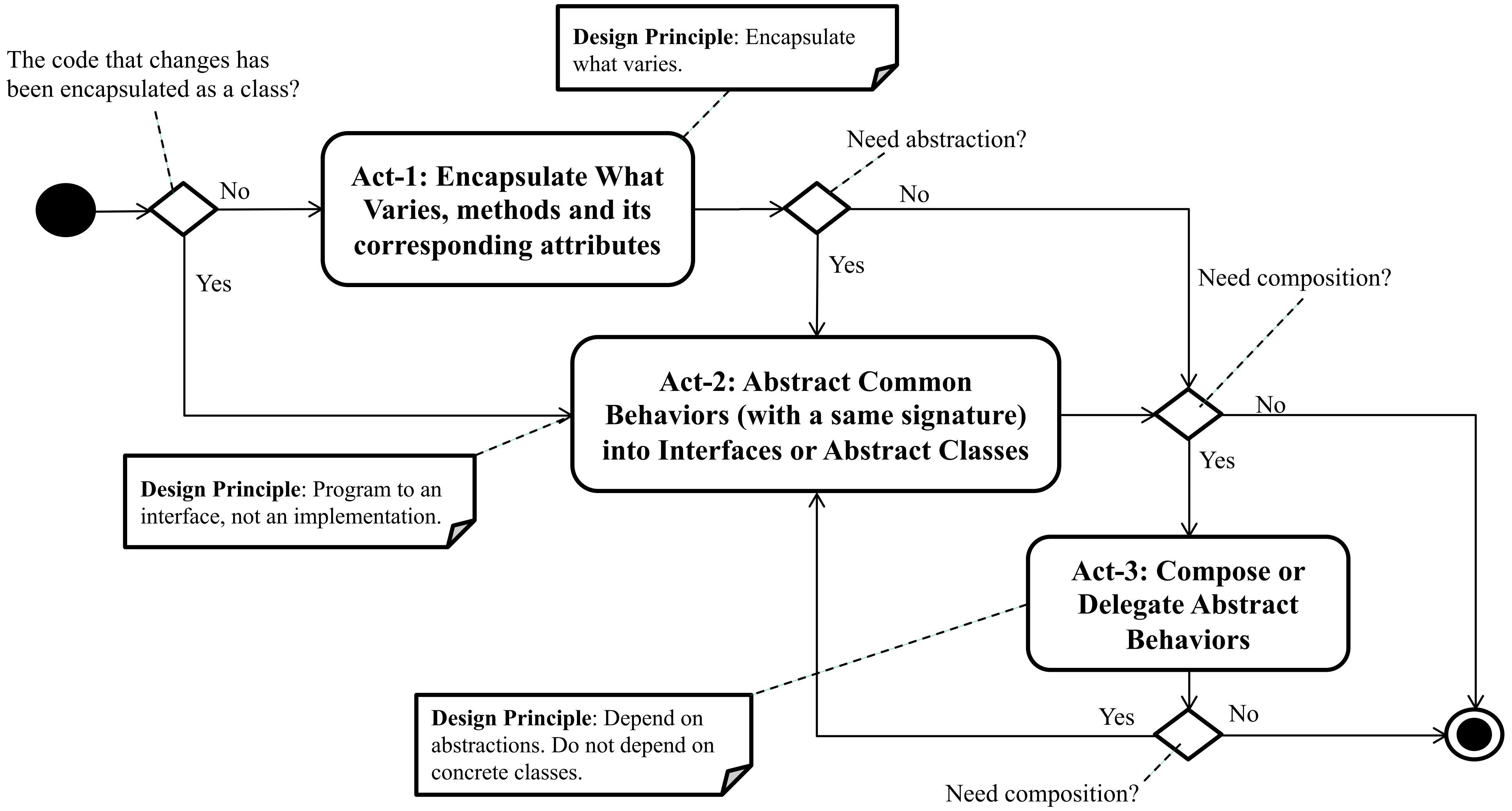
Problems with Initial Design



```
for(int i=0; i<text.size();i++)
        text.get(i).draw();
for(int i=0; i<line.size();i++)
        line.get(i).draw();
for(int i=0; i<rectange.size();i++)
        rectange.get(i).draw();
for(int i=0; i<group.size();i++)
        group.get(i).draw();
```

Problem : If a new requirement is to add a new basic component such as Triangle, then we need to modify Group to meet the new requirement.

Design Process for Change

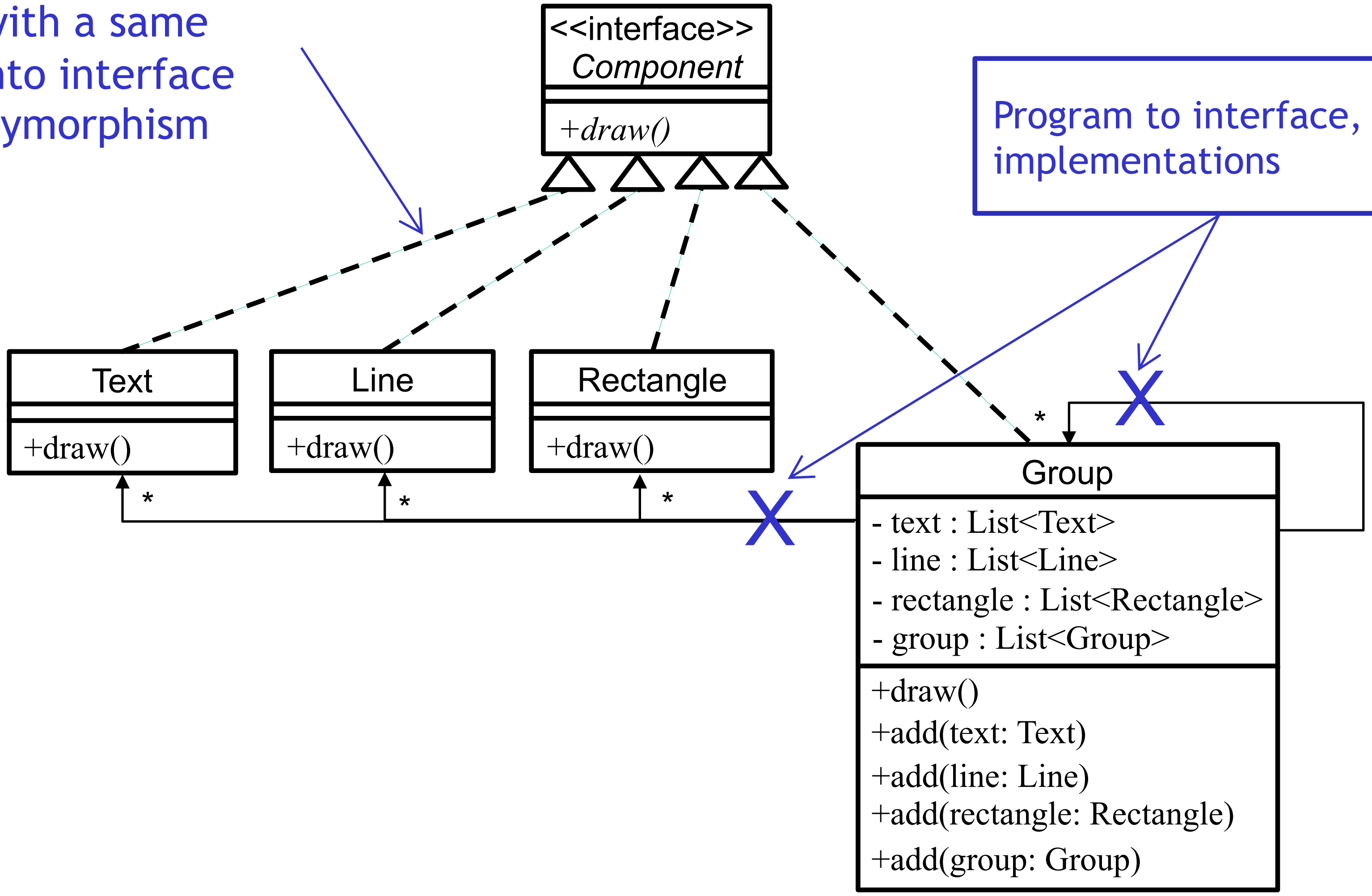


Design Principle: Encapsulate what varies.
The code that changes has been encapsulated as a class?
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

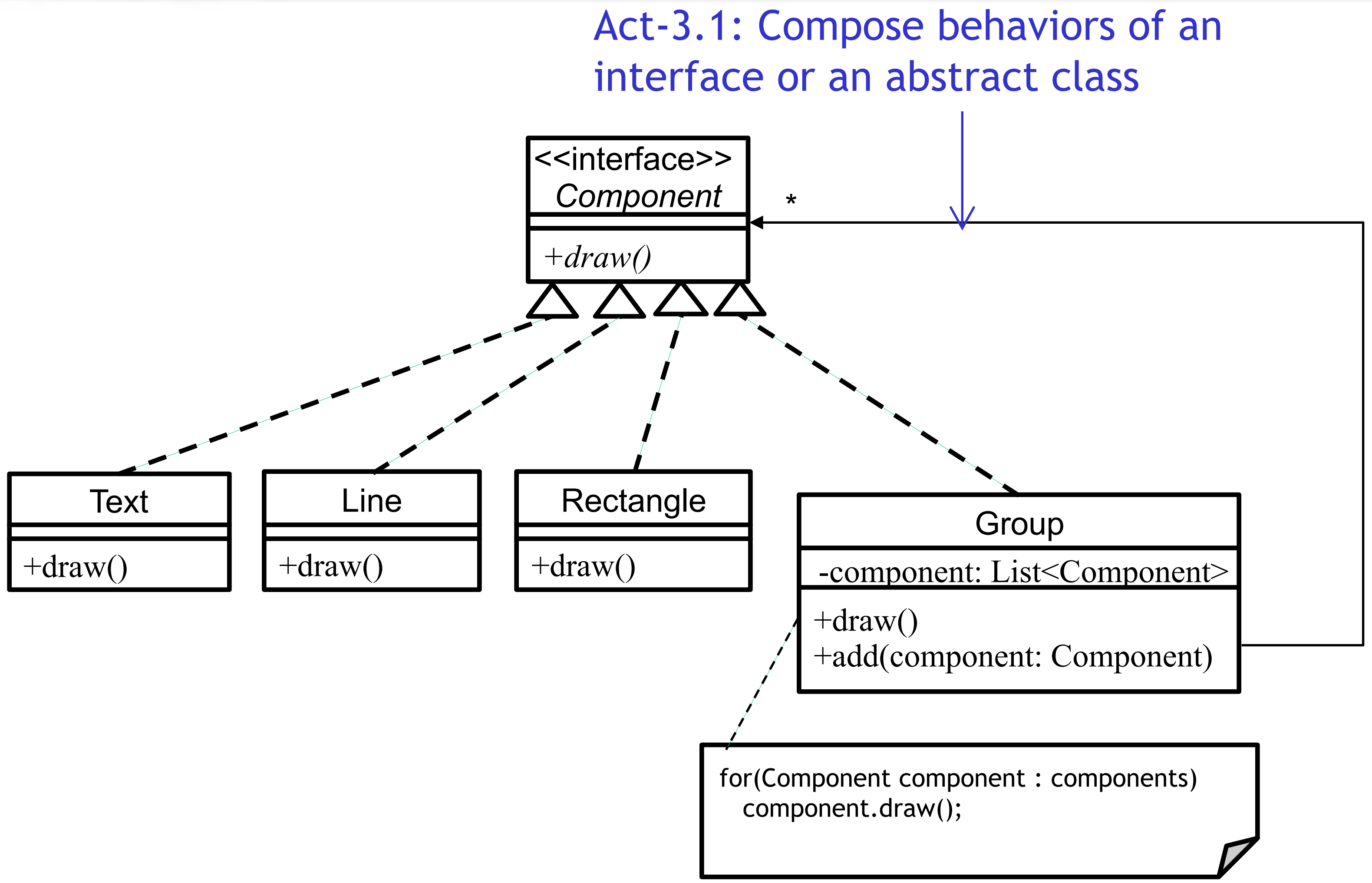
Act-2: Abstract Common Behaviors

Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

Program to interface, not implementations



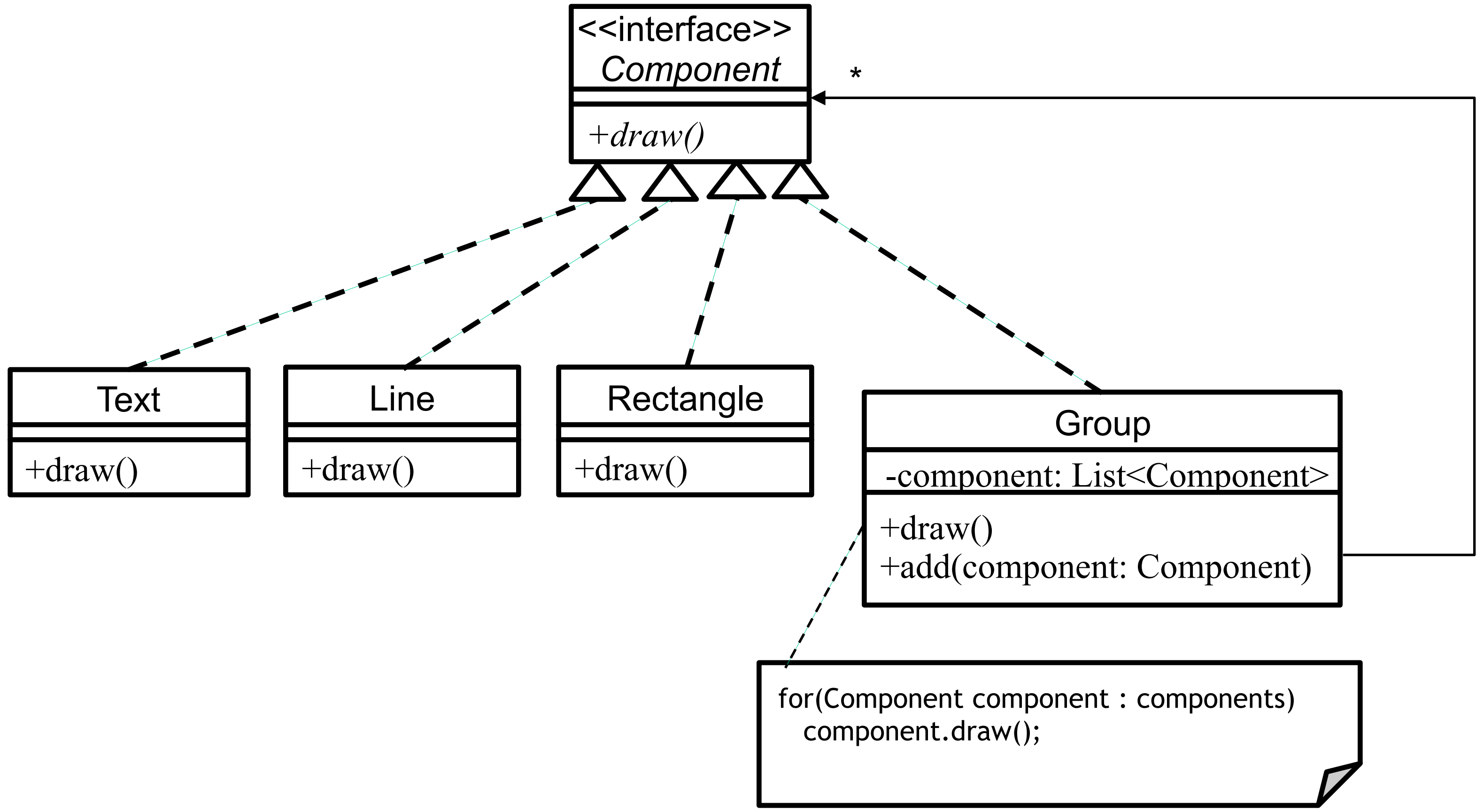
Act-3: Compose Abstract Behaviors



Act-3.1: Compose behaviors of an interface or an abstract class
<<interface>>
Component
+draw()
*
Text
+draw()
Line
+draw()
Rectangle
+draw()
Group
-component: List<Component>
+draw()
+add(component: Component)

```
for(Component component : components)
   component.draw();
```

Refactored Design after Design Process



```
for(Component component : components)
   component.draw();
```

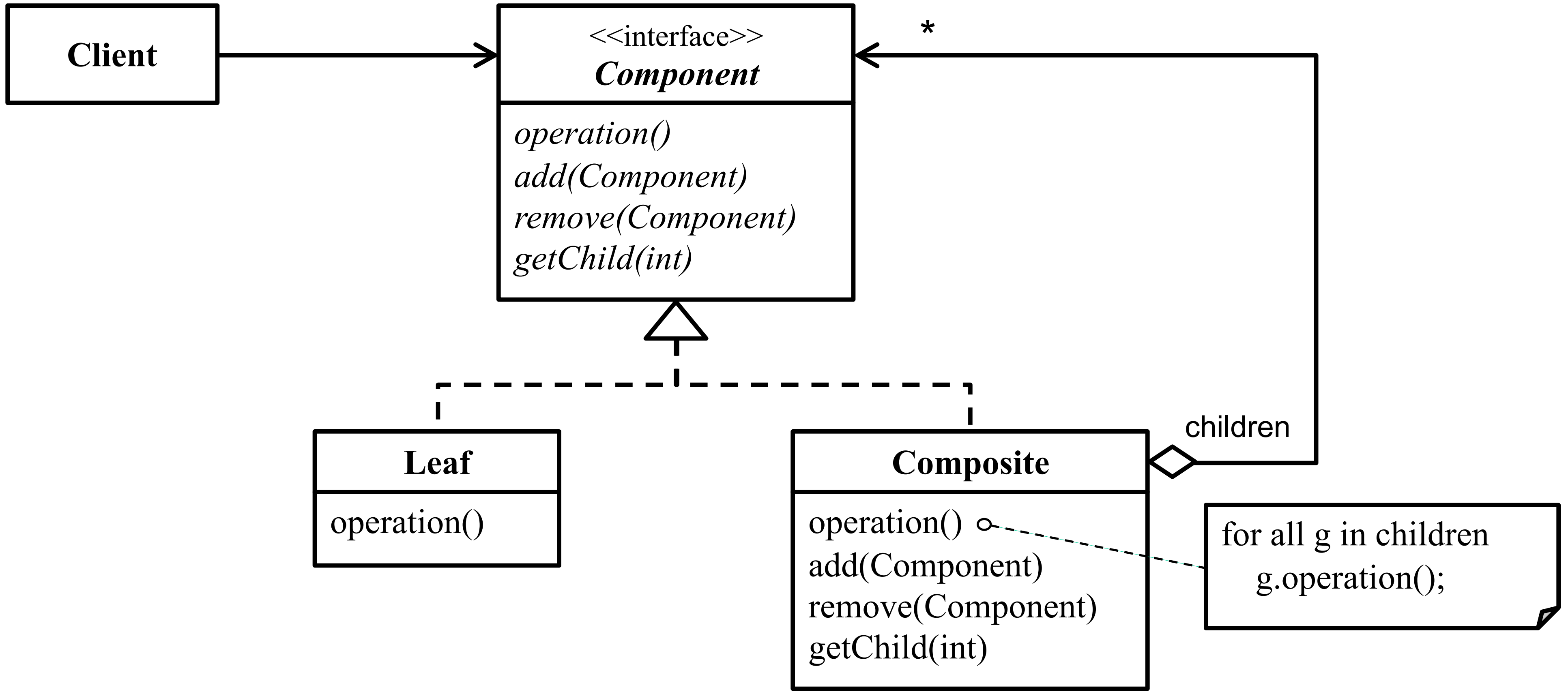
Recurrent Problem

- The user can group components to form larger components, which in turn can be grouped to form still larger components.
  - A simple implementation could define classes for primitives that act as containers for these primitives.
  - But there's a problem with this approach: Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically.

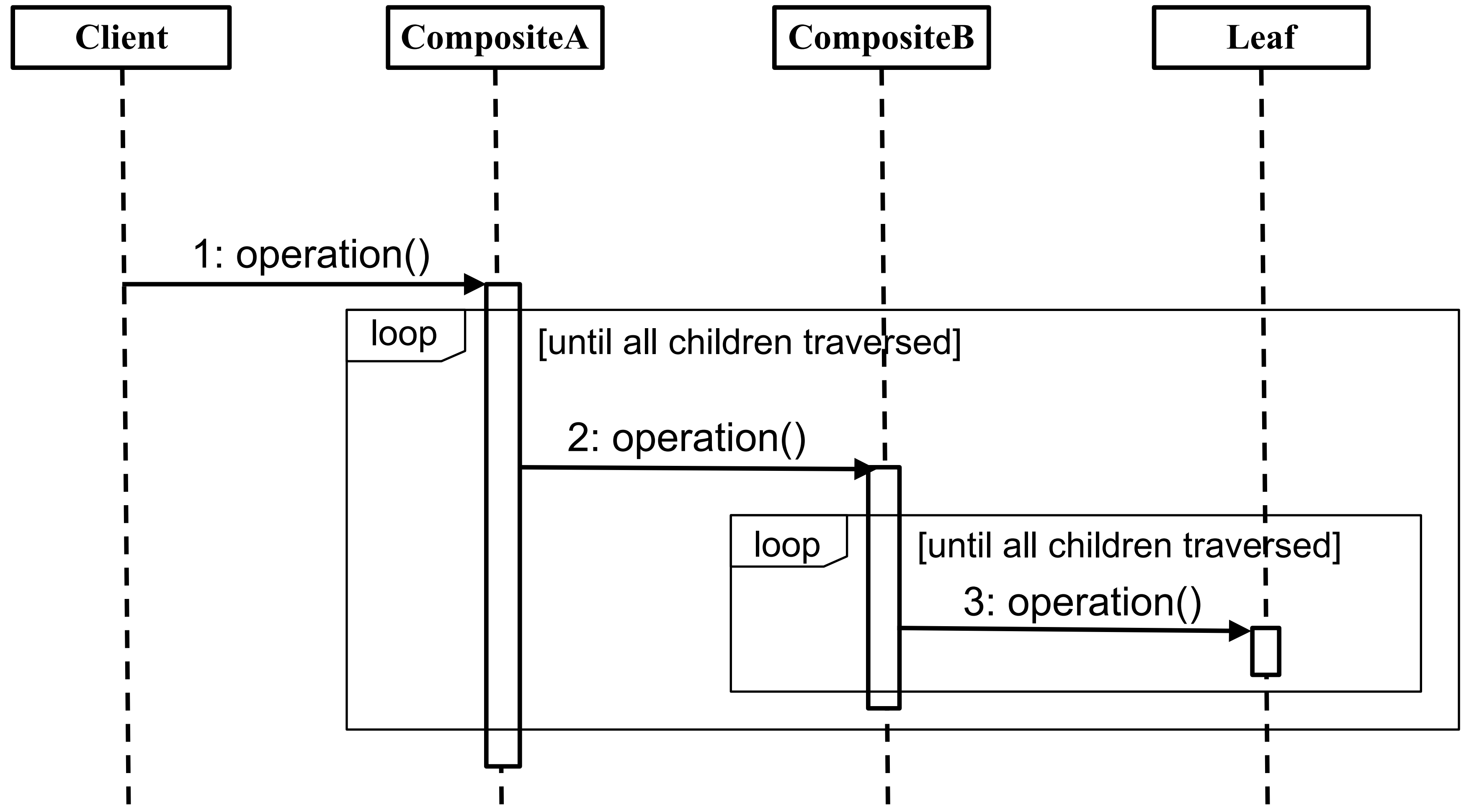
Intent

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

Composite Pattern Structure$_1$



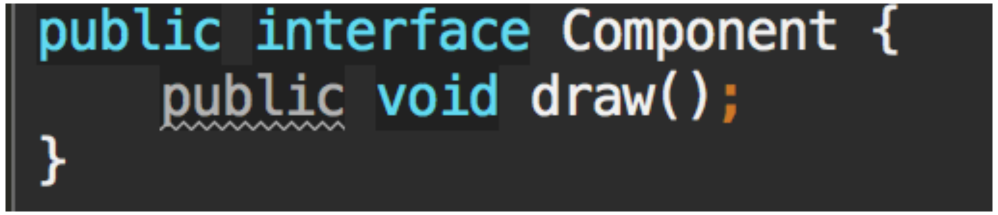
Composite Pattern Structure$_2$



Composite Pattern Structure$_3$

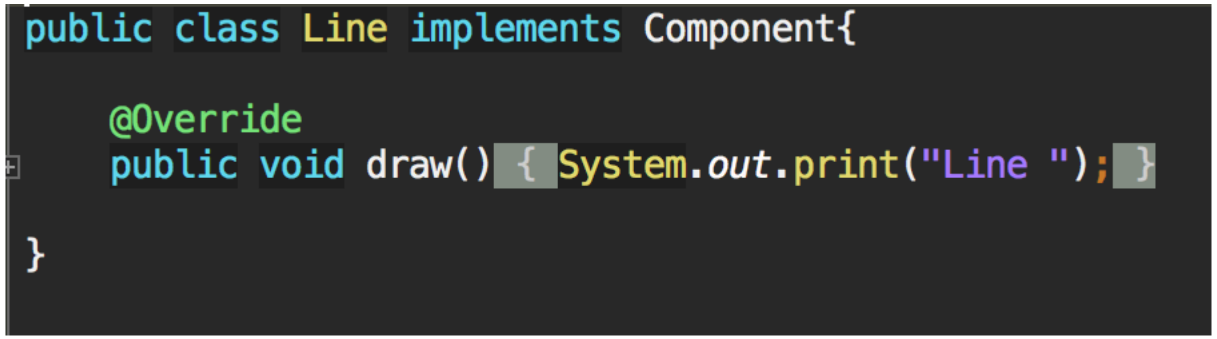
	Instantiation	Use	Termination
Component	X	Client uses this interface to manipulate a Composite class or a Leaf class.	X
Composite	Don’t Care	Client adds, removes, and gets Composite or Leaf objects through Composite who acts as a container. When Client invokes Composite’s operation method, Composite invokes the same method of its child Component objects iteratively.	Don’t Care
Leaf	Don’t Care	Client adds, removes, and gets Leaf objects to/from Composite. Leaf executes its operation method when Composite or Client requests through polymorphism.	Don’t Care

Component



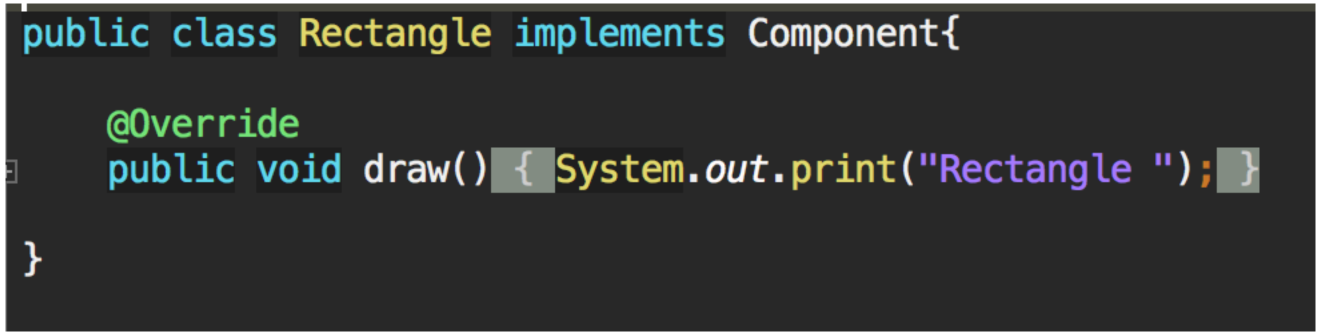
```java
public interface Component {
    public void draw();
}
```

Line



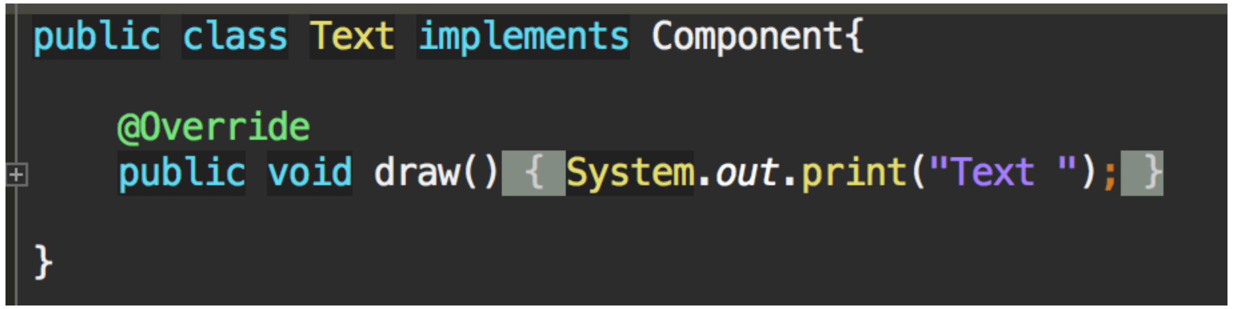
```
public class Line implements Component{

    @Override
    public void draw() { System.out.print("Line "); }

}
```

Rectangle



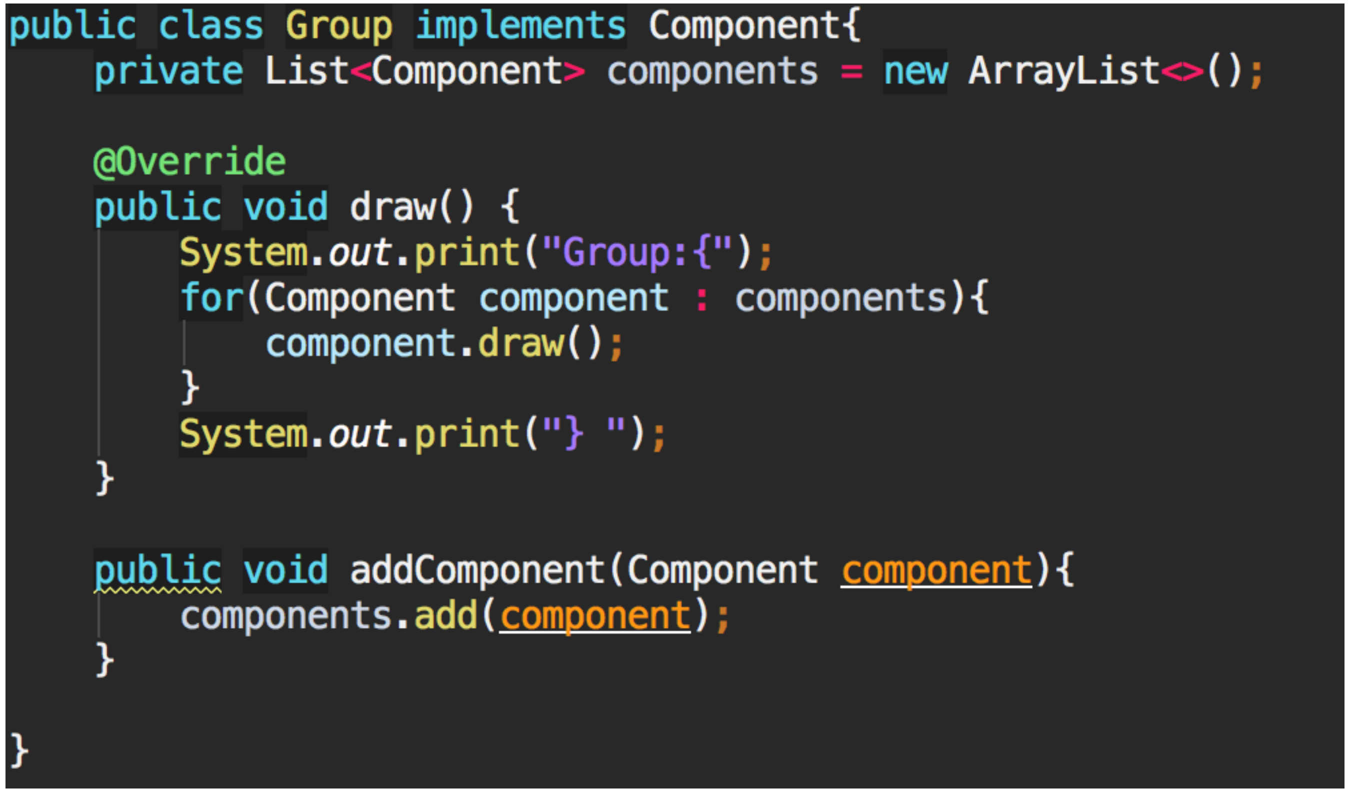
```
public class Rectangle implements Component{

    @Override
    public void draw() { System.out.print("Rectangle "); }

}
```

Text



```
public class Text implements Component{

    @Override
    public void draw() { System.out.print("Text "); }

}
```

Group



```
public class Group implements Component{
    private List<Component> components = new ArrayList<>();

    @Override
    public void draw() {
        System.out.print("Group:{");
        for(Component component : components){
            component.draw();
        }
        System.out.print("} ");
    }

    public void addComponent(Component component){
        components.add(component);
    }

}
```

Input / Output

Input:

```
//The input file format is XML

<?xml version="1.0"?>

<Question>

    <[Basic_Component]/>

    …

    [Larger_Component]

    …

</Question>
```

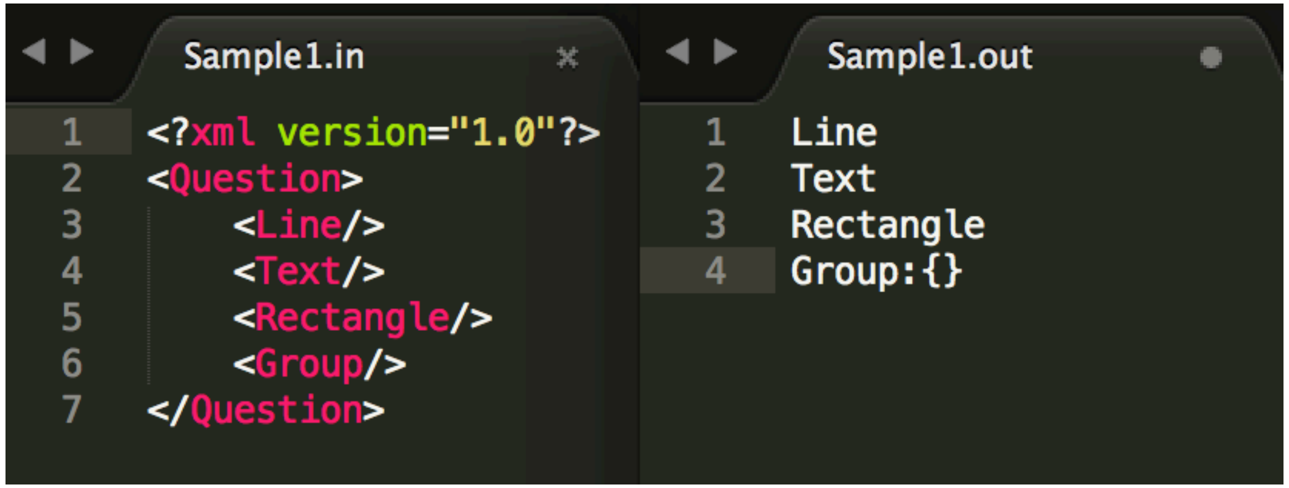
Output:

```
[Basic_Component]

…

Group:{[Basic_Component]…}

…
```

Test cases

❑ TestCase 1:	4 kinds of Component
❑ TestCase 2:	Group compose other
❑ TestCase 3:	Complex

Test case1



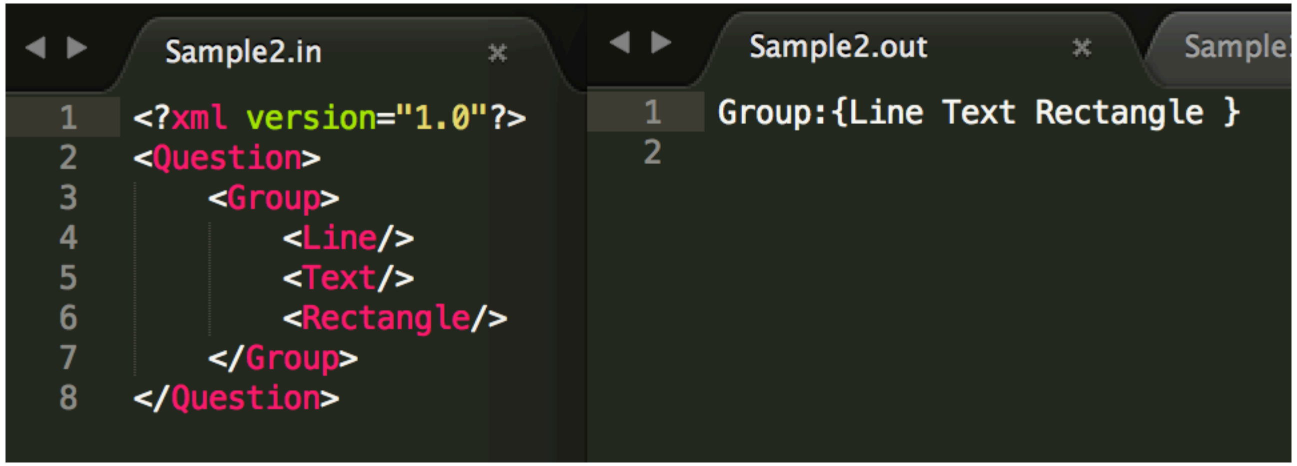
Sample1.in

```
<?xml version="1.0"?>
<Question>
    <Line/>
    <Text/>
    <Rectangle/>
    <Group/>
</Question>
```

Sample1.out

```
Line
Text
Rectangle
Group:{}
```

Test case2



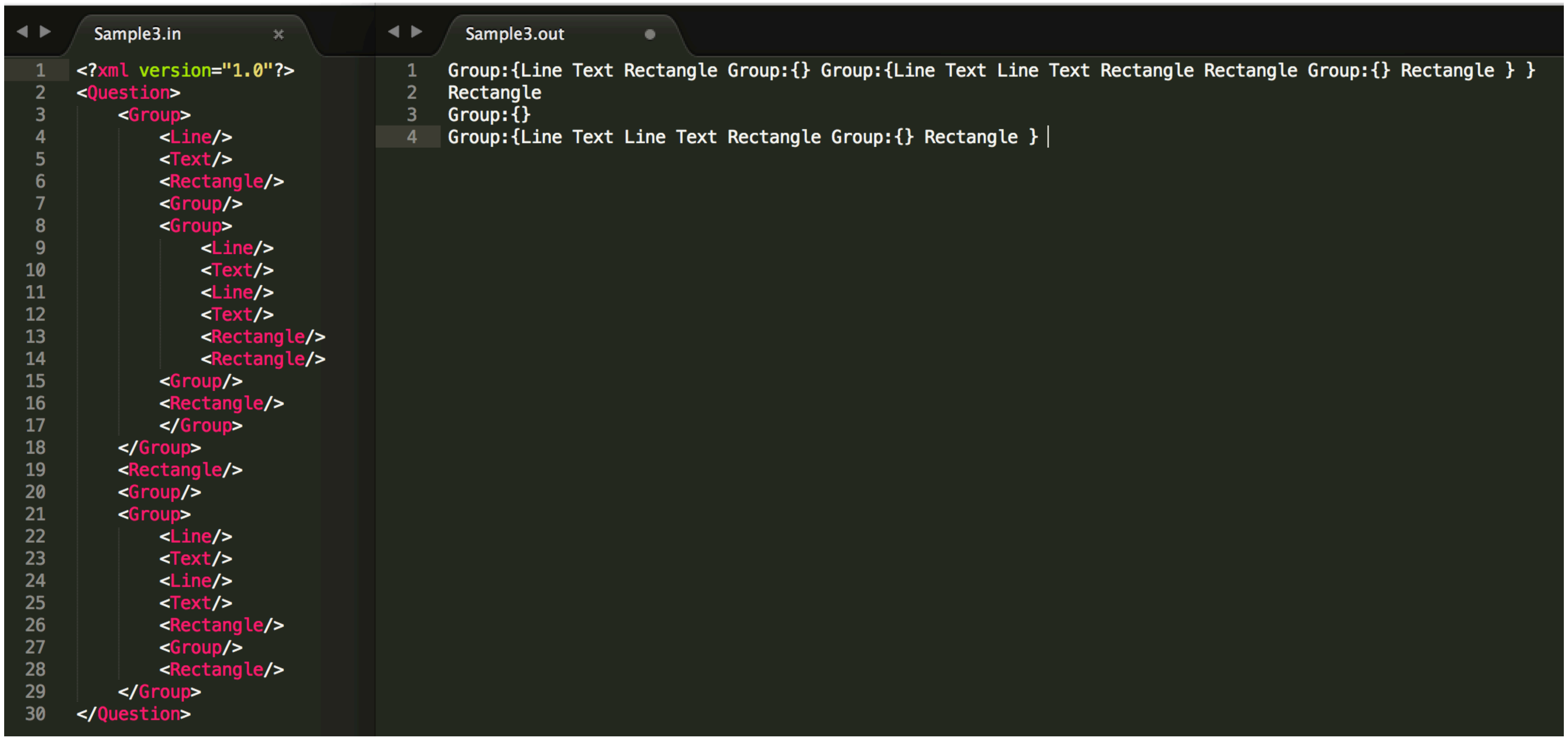
Sample2.in

```
<?xml version="1.0"?>
<Question>
    <Group>
        <Line/>
        <Text/>
        <Rectangle/>
    </Group>
</Question>
```

Sample2.out

```
Group:{Line Text Rectangle }
```

Test case3



Sample3.in

```xml
<?xml version="1.0"?>
<Question>
    <Group>
        <Line/>
        <Text/>
        <Rectangle/>
        <Group/>
        <Group>
            <Line/>
            <Text/>
            <Line/>
            <Text/>
            <Rectangle/>
            <Rectangle/>
        <Group/>
        <Rectangle/>
        </Group>
    </Group>
    <Rectangle/>
    <Group/>
    <Group>
        <Line/>
        <Text/>
        <Line/>
        <Text/>
        <Rectangle/>
        <Group/>
        <Rectangle/>
    </Group>
</Question>
```

Sample3.out

```
Group:{Line Text Rectangle Group:{} Group:{Line Text Line Text Rectangle Rectangle Group:{} Rectangle } }
Rectangle
Group:{}
Group:{Line Text Line Text Rectangle Group:{} Rectangle }
```

Extended Merge of Two Menus (Composite)

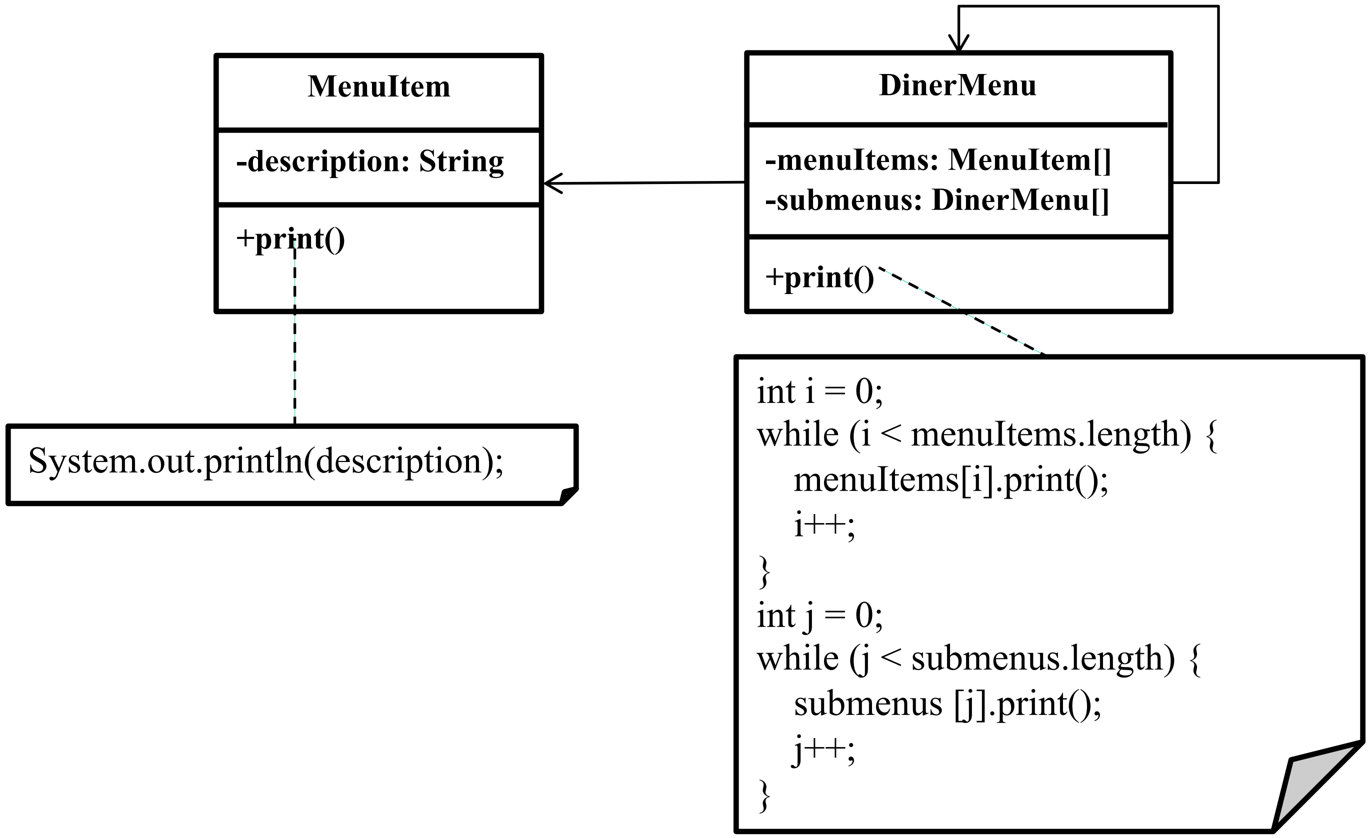
Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statement

- Based on the Merge Two Menus example,
  - A waitress of Pancake House keeps a breakfast menu which uses an ArrayList to hold its menu items.
  - And a waitress of Diner keeps a lunch menu which uses an array to hold its menu items.
  - Now, these two restaurants are merged and intend to provide service in one place, so a waitress should keep both menus in hands.
  - The waitress would like to print two different menu representations at a time.
- A dessert submenu is added to the Diner menu.

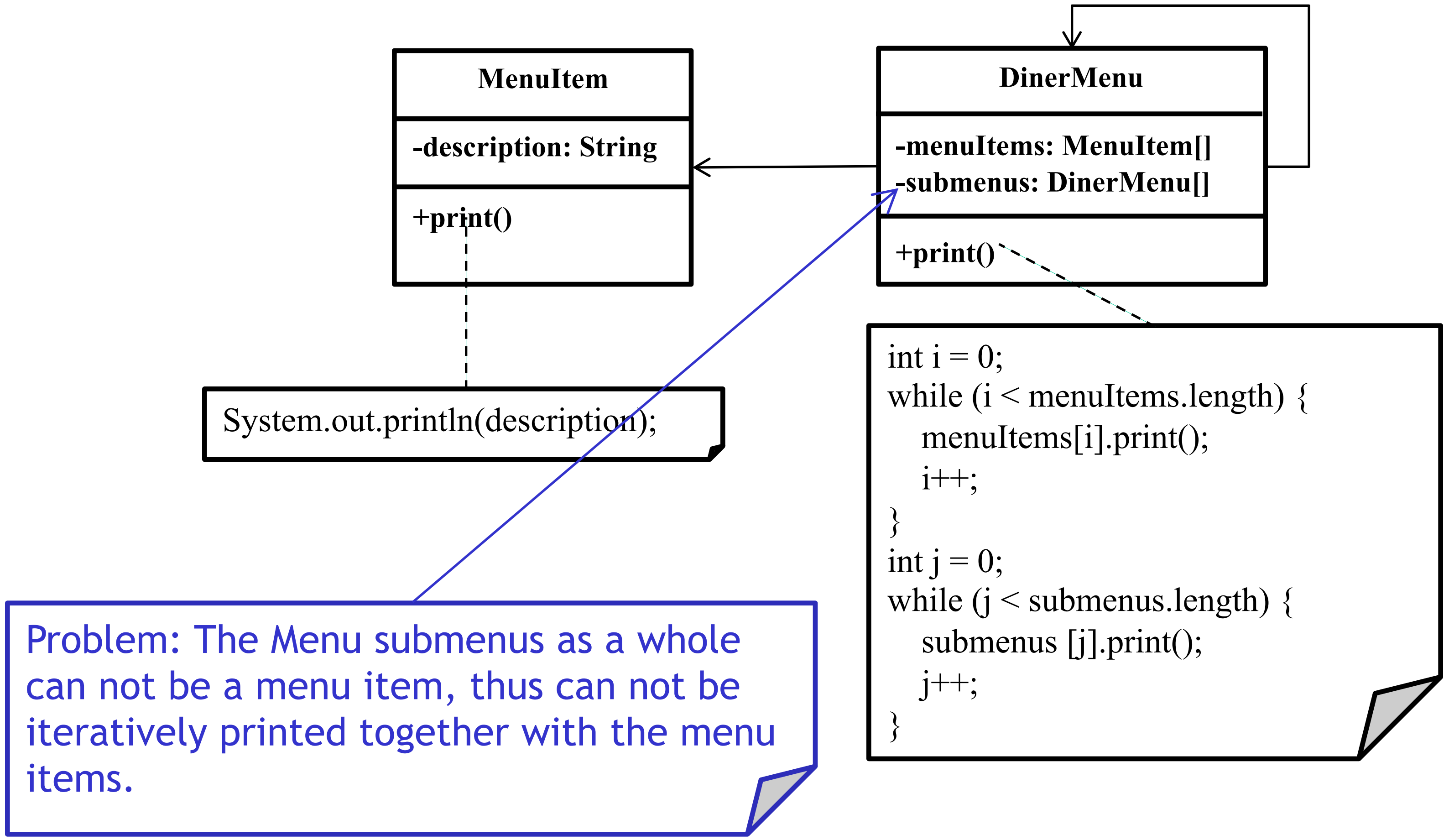
Initial Design - Class Diagram



```
System.out.println(description);
```

```
int i = 0;
while (i < menuItems.length) {
   menuItems[i].print();
   i++;
}
int j = 0;
while (j < submenus.length) {
   submenus [j].print();
   j++;
}
```

Problems with Initial Design

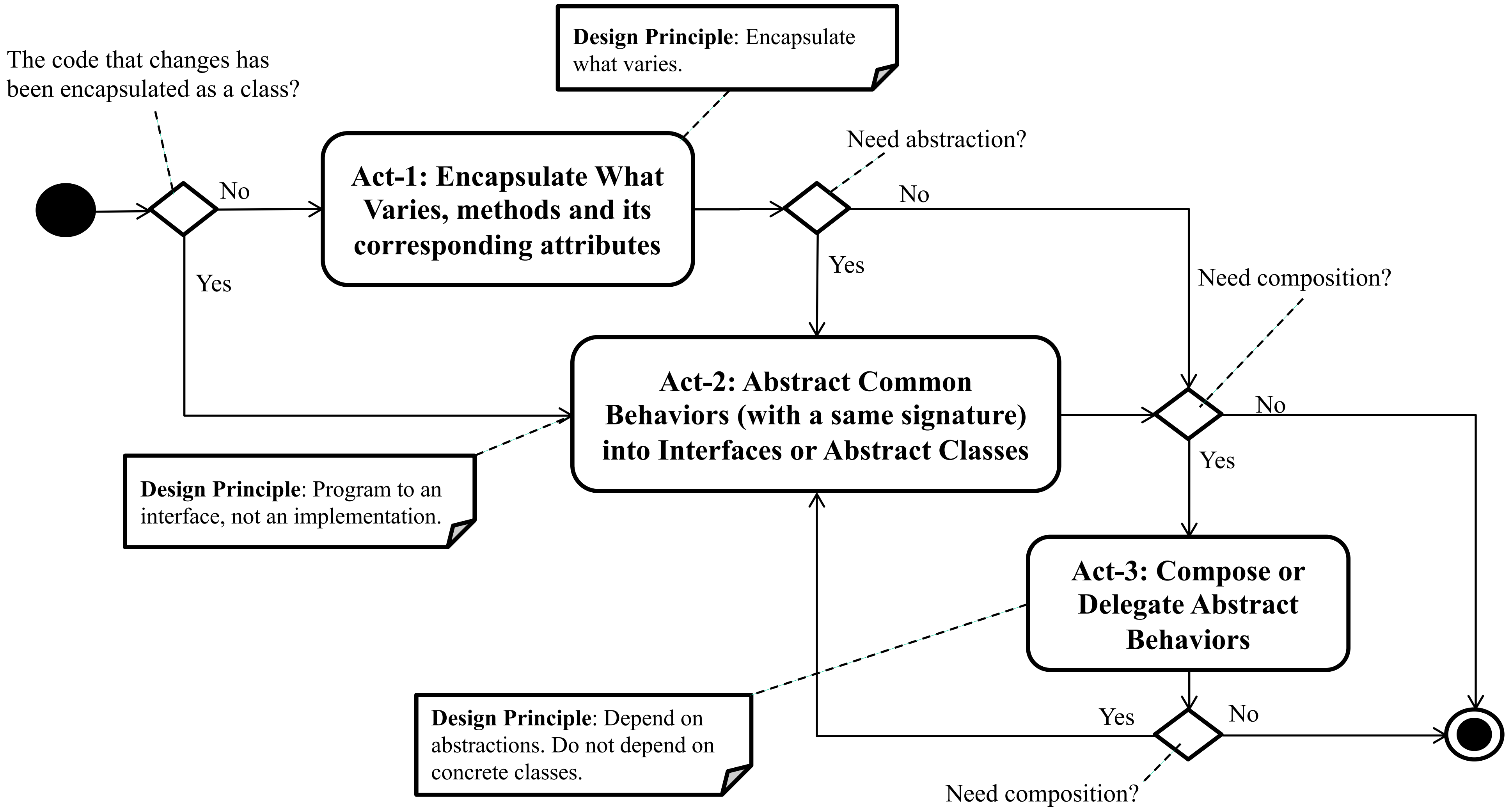


```
System.out.println(description);
```

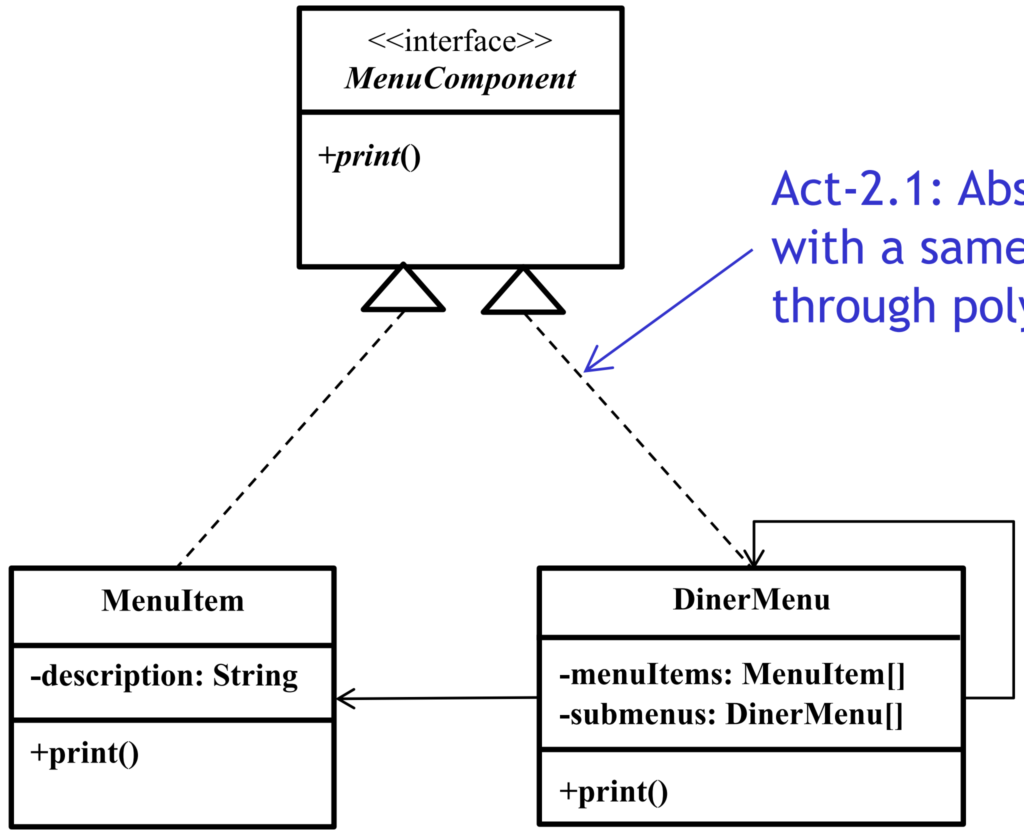
```
int i = 0;
while (i < menuItems.length) {
   menuItems[i].print();
   i++;
}
int j = 0;
while (j < submenus.length) {
   submenus [j].print();
   j++;
}
```

Problem: The Menu submenus as a whole can not be a menu item, thus can not be iteratively printed together with the menu items.

Design Process for Change

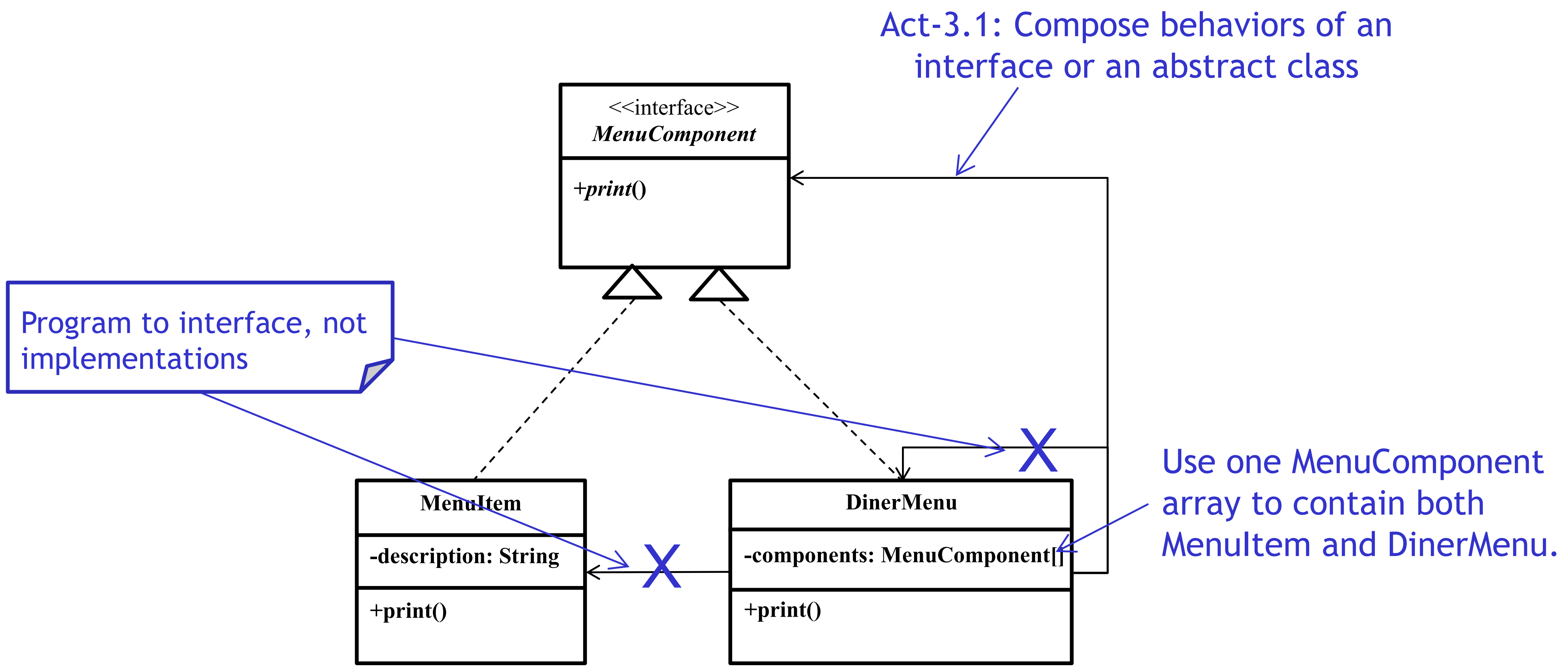

Design Principle: Encapsulate what varies.
The code that changes has been encapsulated as a class?
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Yes
No
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Need composition?

Act-2: Abstract Common Behaviors



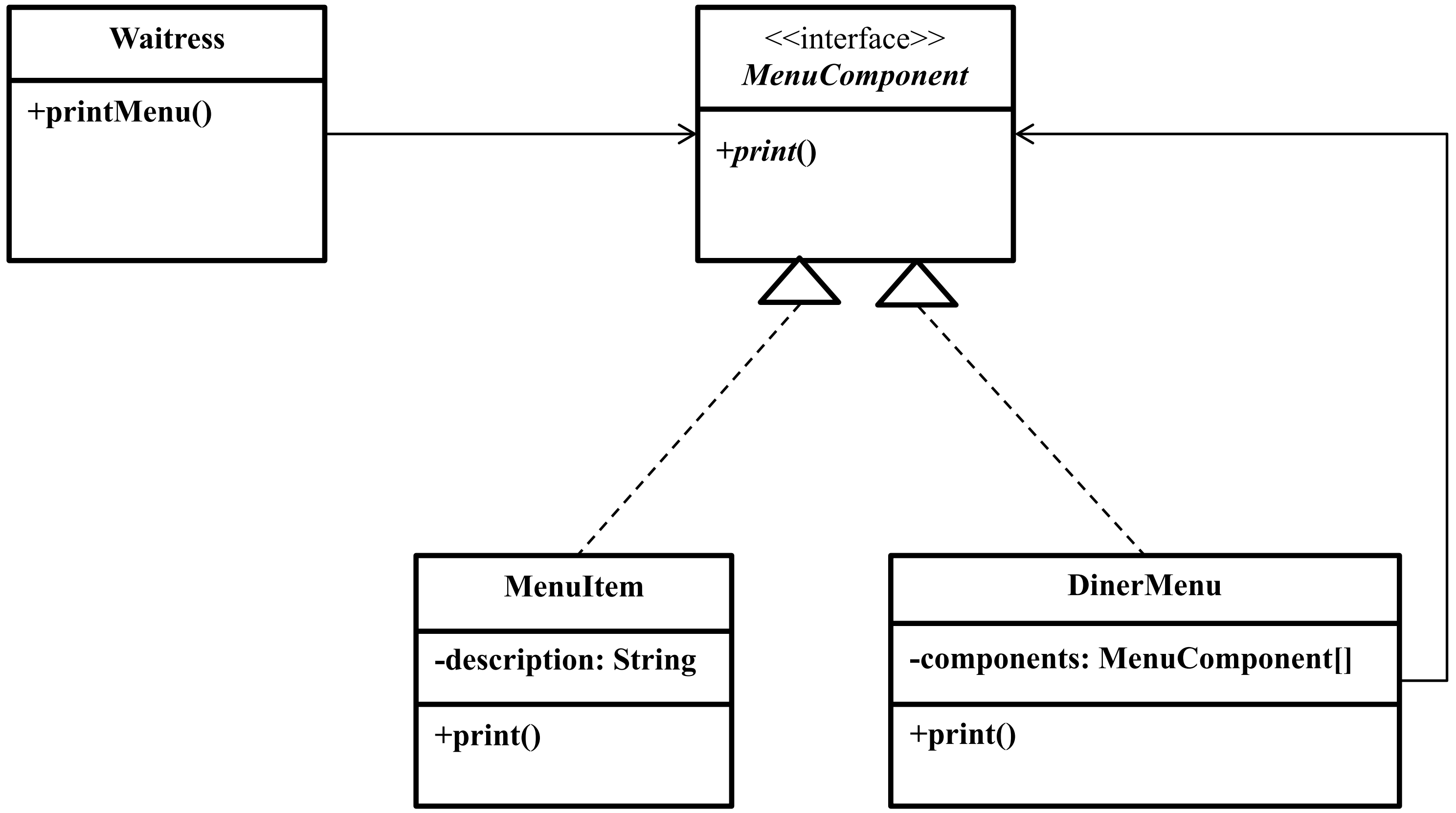
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

Act-3: Compose Abstract Behaviors

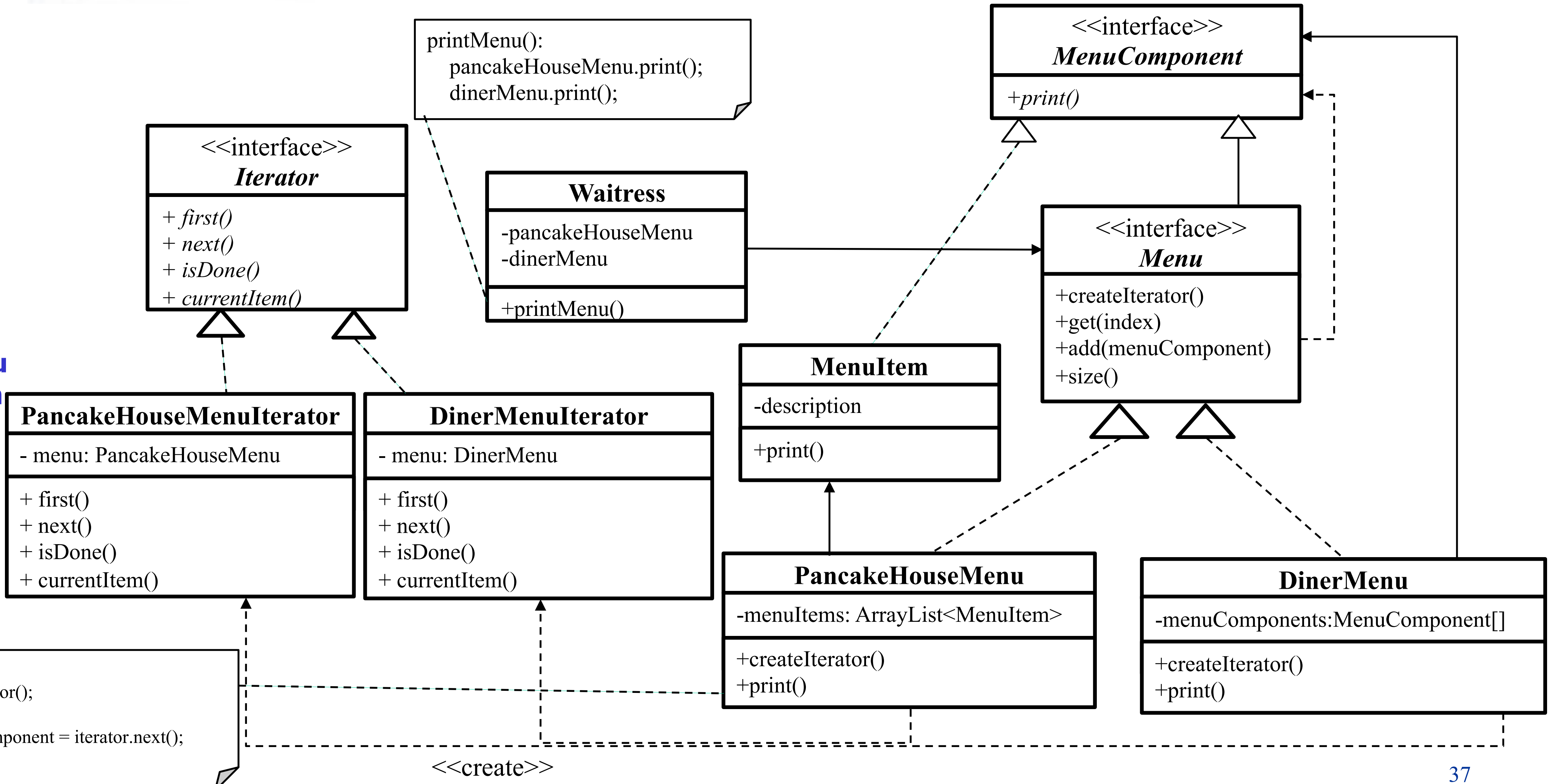


Act-3.1: Compose behaviors of an interface or an abstract class
<<interface>>
MenuComponent
+print()
Program to interface, not implementations
X
X
MenuItem
-description: String
+print()
DinerMenu
-components: MenuComponent[]
+print()
Use one MenuComponent array to contain both MenuItem and DinerMenu.

Refactored Design after Design Process



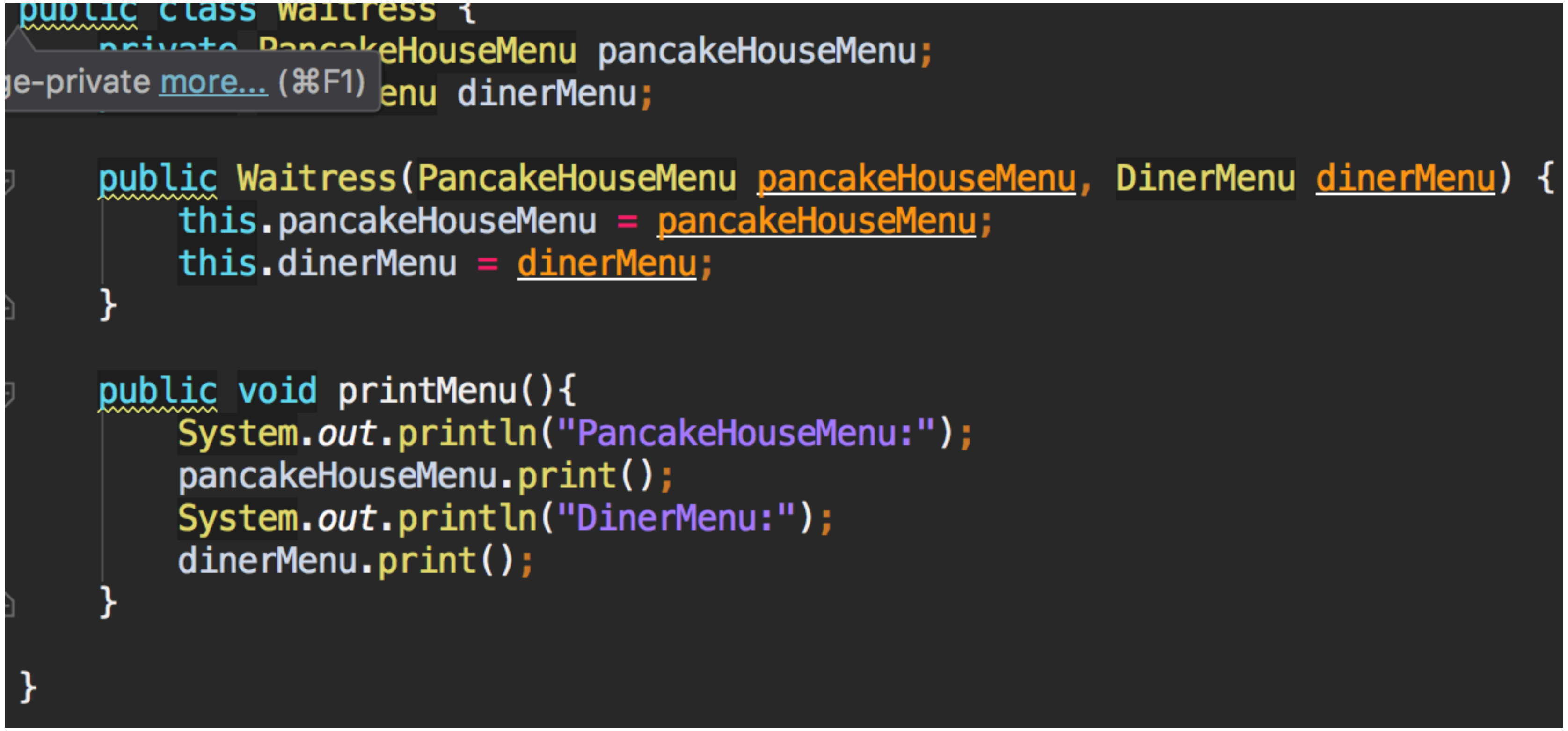
Print Menus in Composite with Iterator



traverses its menu components by an iterator, and then prints them out.

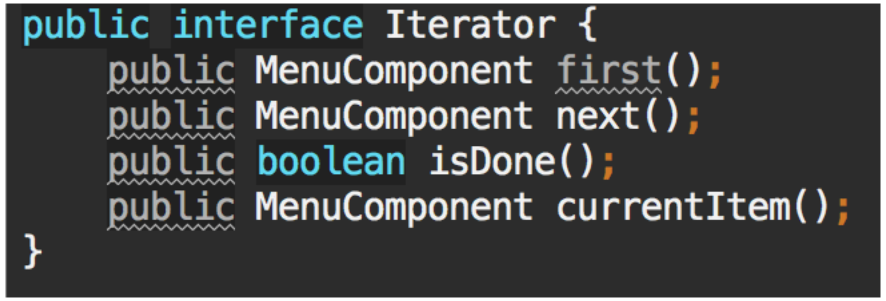
```
print():
  Iterator iterator = createIterator();
  while(!iterator.isDone()){
    MenuComponent menuComponent = iterator.next();
    menuComponent.print();
  }
```

Waitress



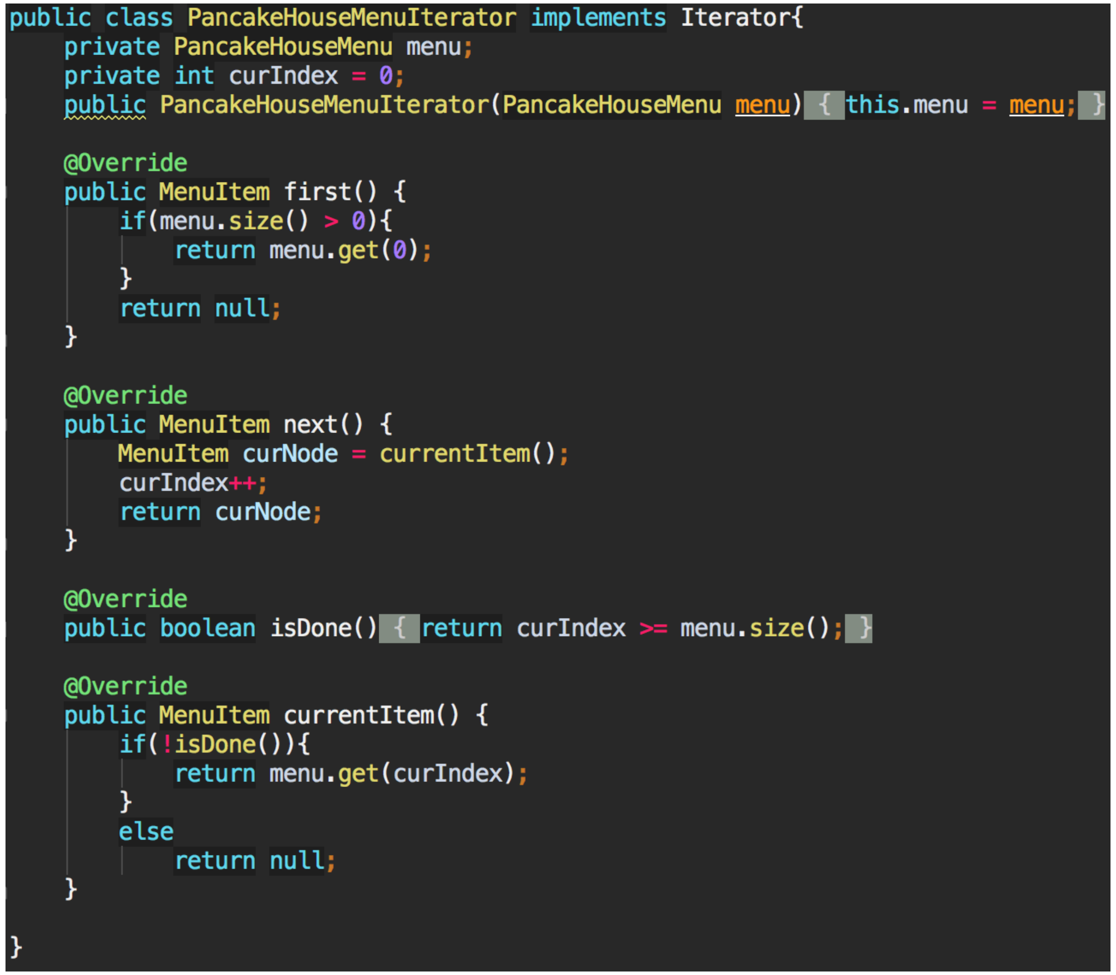
```
public class Waitress {
    private PancakeHouseMenu pancakeHouseMenu;
ge-private more... (⌘F1) enu dinerMenu;

    public Waitress(PancakeHouseMenu pancakeHouseMenu, DinerMenu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu;
        this.dinerMenu = dinerMenu;
    }

    public void printMenu(){
        System.out.println("PancakeHouseMenu:");
        pancakeHouseMenu.print();
        System.out.println("DinerMenu:");
        dinerMenu.print();
    }

}
```

Iterator



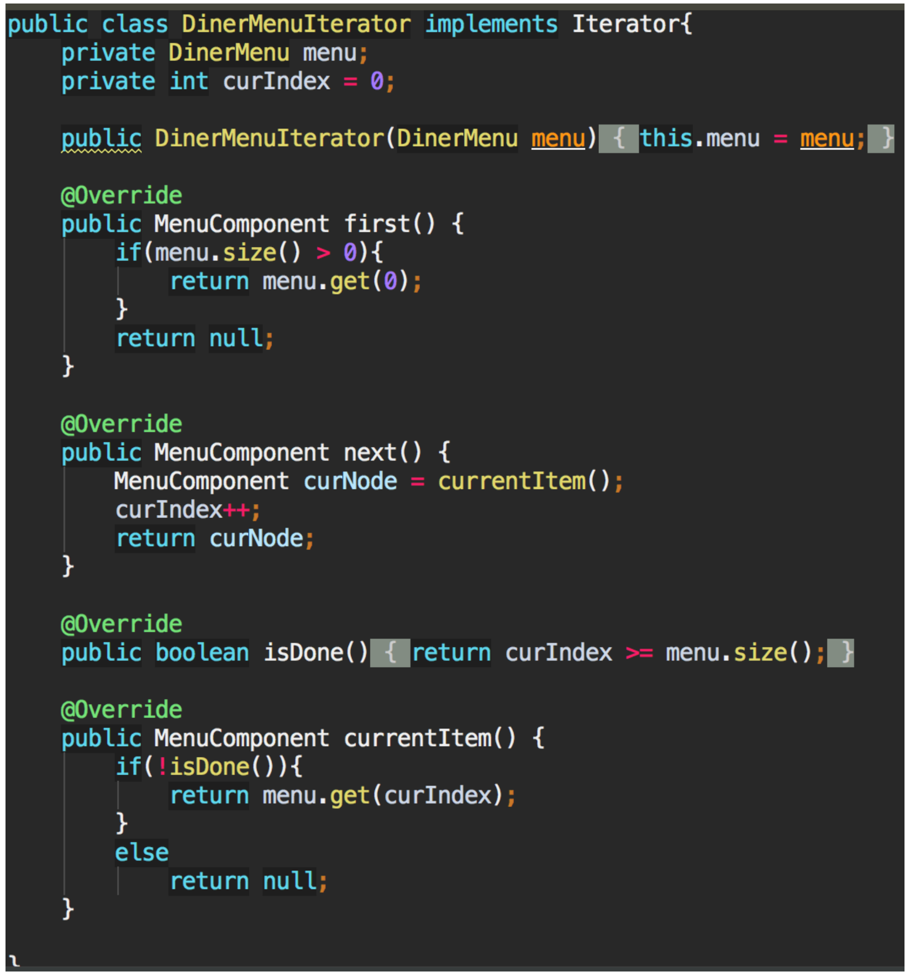
```
public interface Iterator {
    public MenuComponent first();
    public MenuComponent next();
    public boolean isDone();
    public MenuComponent currentItem();
}
```

PancakeHouseIterator



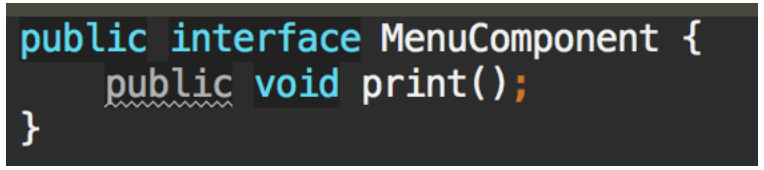
```java
public class PancakeHouseMenuIterator implements Iterator{
    private PancakeHouseMenu menu;
    private int curIndex = 0;
    public PancakeHouseMenuIterator(PancakeHouseMenu menu) { this.menu = menu; }

    @Override
    public MenuItem first() {
        if(menu.size() > 0){
            return menu.get(0);
        }
        return null;
    }

    @Override
    public MenuItem next() {
        MenuItem curNode = currentItem();
        curIndex++;
        return curNode;
    }

    @Override
    public boolean isDone() { return curIndex >= menu.size(); }

    @Override
    public MenuItem currentItem() {
        if(!isDone()){
            return menu.get(curIndex);
        }
        else
            return null;
    }

}
```

DinerMenuIterator



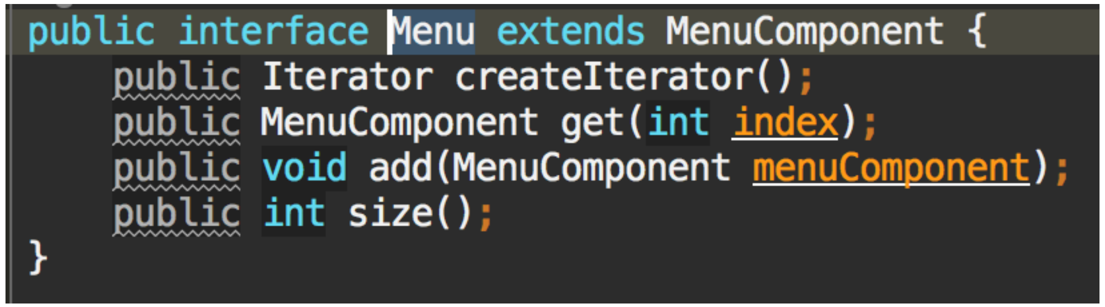
```java
public class DinerMenuIterator implements Iterator{
    private DinerMenu menu;
    private int curIndex = 0;

    public DinerMenuIterator(DinerMenu menu) { this.menu = menu; }

    @Override
    public MenuComponent first() {
        if(menu.size() > 0){
            return menu.get(0);
        }
        return null;
    }

    @Override
    public MenuComponent next() {
        MenuComponent curNode = currentItem();
        curIndex++;
        return curNode;
    }

    @Override
    public boolean isDone() { return curIndex >= menu.size(); }

    @Override
    public MenuComponent currentItem() {
        if(!isDone()){
            return menu.get(curIndex);
        }
        else
            return null;
    }
```

MenuComponent



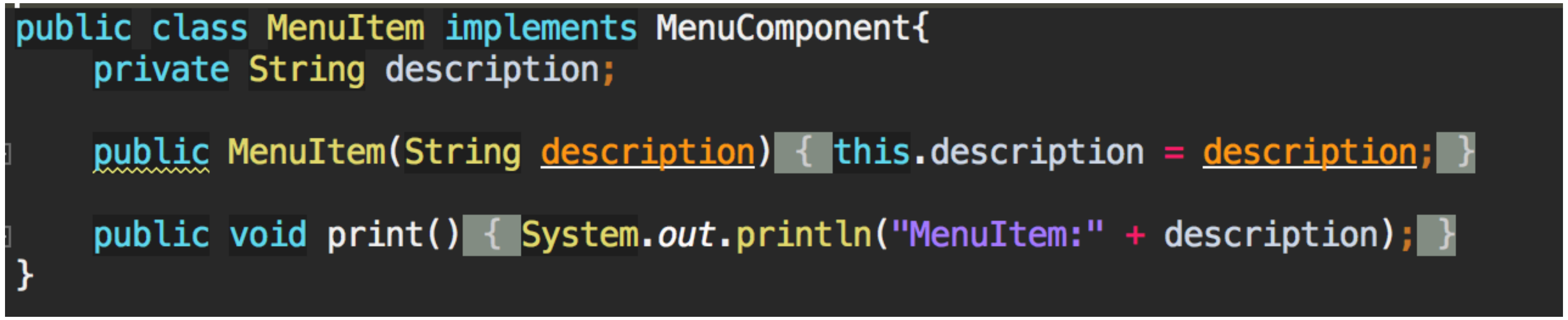
```
public interface MenuComponent {
    public void print();
}
```

Menu



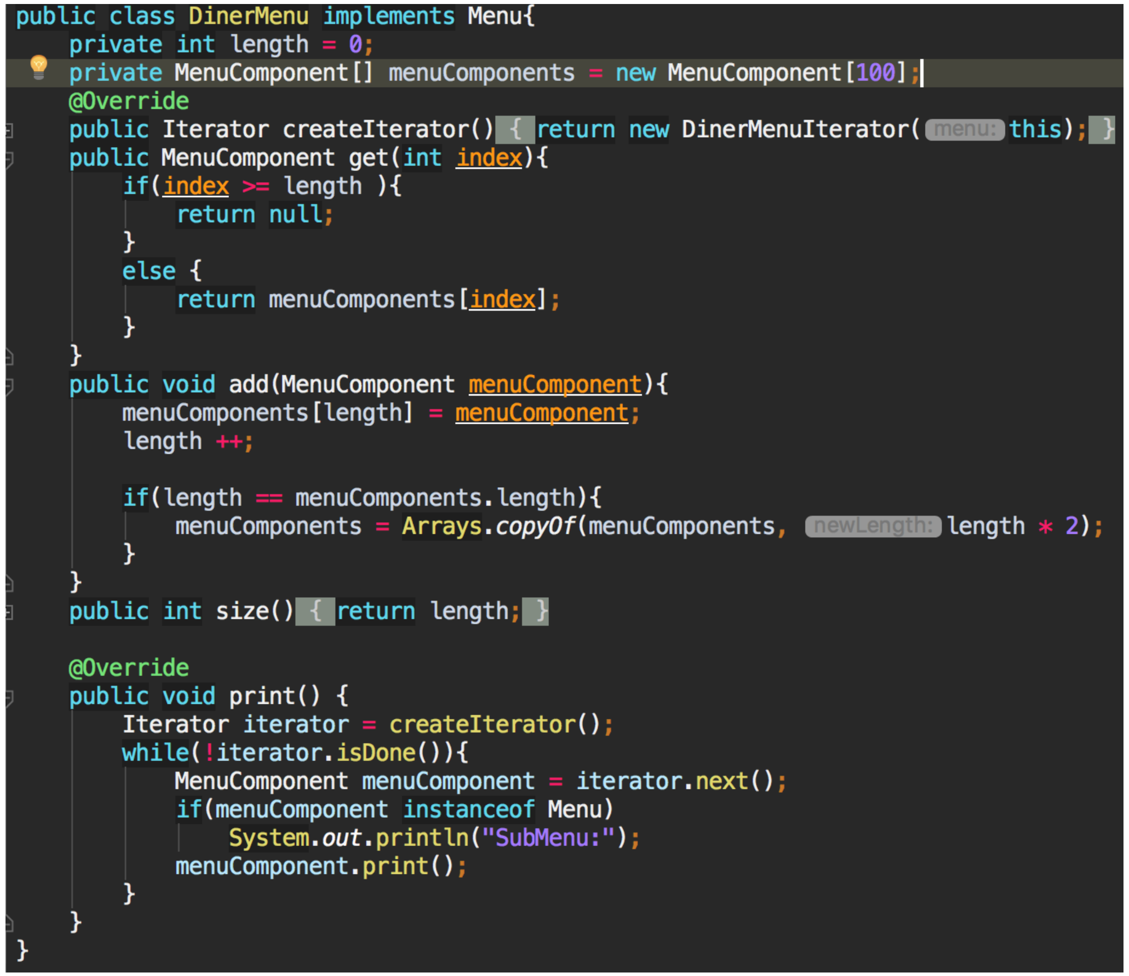
```
public interface Menu extends MenuComponent {
    public Iterator createIterator();
    public MenuComponent get(int index);
    public void add(MenuComponent menuComponent);
    public int size();
}
```

MenuItem



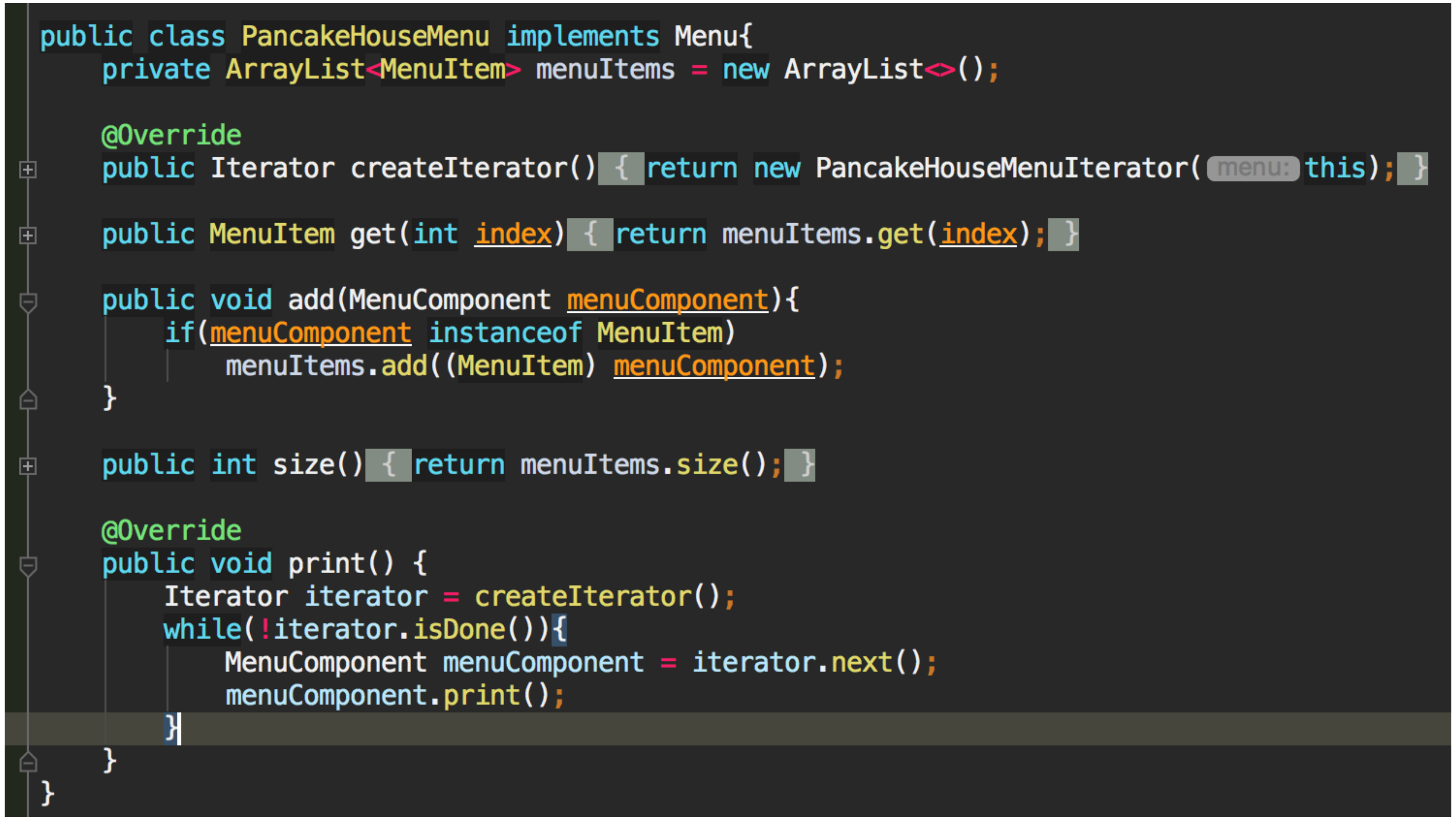
```
public class MenuItem implements MenuComponent{
    private String description;

    public MenuItem(String description) { this.description = description; }

    public void print() { System.out.println("MenuItem:" + description); }
}
```

DinerMenu



```
public class DinerMenu implements Menu{
    private int length = 0;
    private MenuComponent[] menuComponents = new MenuComponent[100];
    @Override
    public Iterator createIterator() { return new DinerMenuIterator( menu: this); }
    public MenuComponent get(int index){
        if(index >= length ){
            return null;
        }
        else {
            return menuComponents[index];
        }
    }
    public void add(MenuComponent menuComponent){
        menuComponents[length] = menuComponent;
        length ++;

        if(length == menuComponents.length){
            menuComponents = Arrays.copyOf(menuComponents,  newLength: length * 2);
        }
    }
    public int size() { return length; }

    @Override
    public void print() {
        Iterator iterator = createIterator();
        while(!iterator.isDone()){
            MenuComponent menuComponent = iterator.next();
            if(menuComponent instanceof Menu)
                System.out.println("SubMenu:");
            menuComponent.print();
        }
    }
}
```

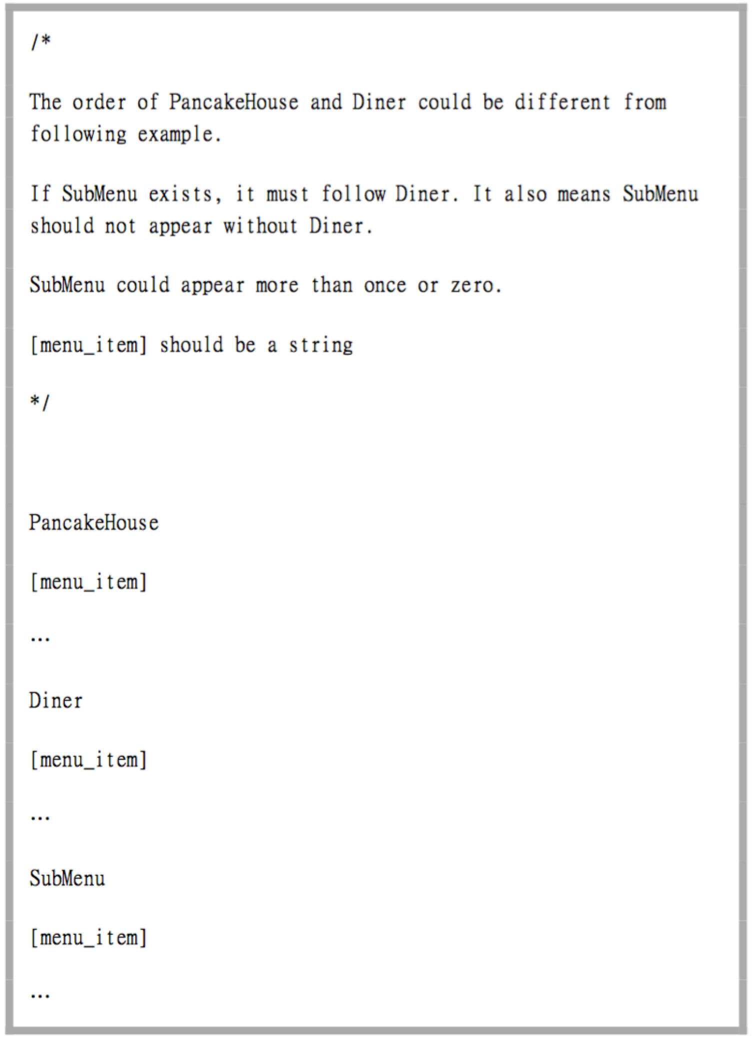
PancakeHouseMenu



```
public class PancakeHouseMenu implements Menu{
    private ArrayList<MenuItem> menuItems = new ArrayList<>();

    @Override
    public Iterator createIterator() { return new PancakeHouseMenuIterator( menu: this); }

    public MenuItem get(int index) { return menuItems.get(index); }

    public void add(MenuComponent menuComponent){
        if(menuComponent instanceof MenuItem)
            menuItems.add((MenuItem) menuComponent);
    }

    public int size() { return menuItems.size(); }

    @Override
    public void print() {
        Iterator iterator = createIterator();
        while(!iterator.isDone()){
            MenuComponent menuComponent = iterator.next();
            menuComponent.print();
        }
    }
}
```

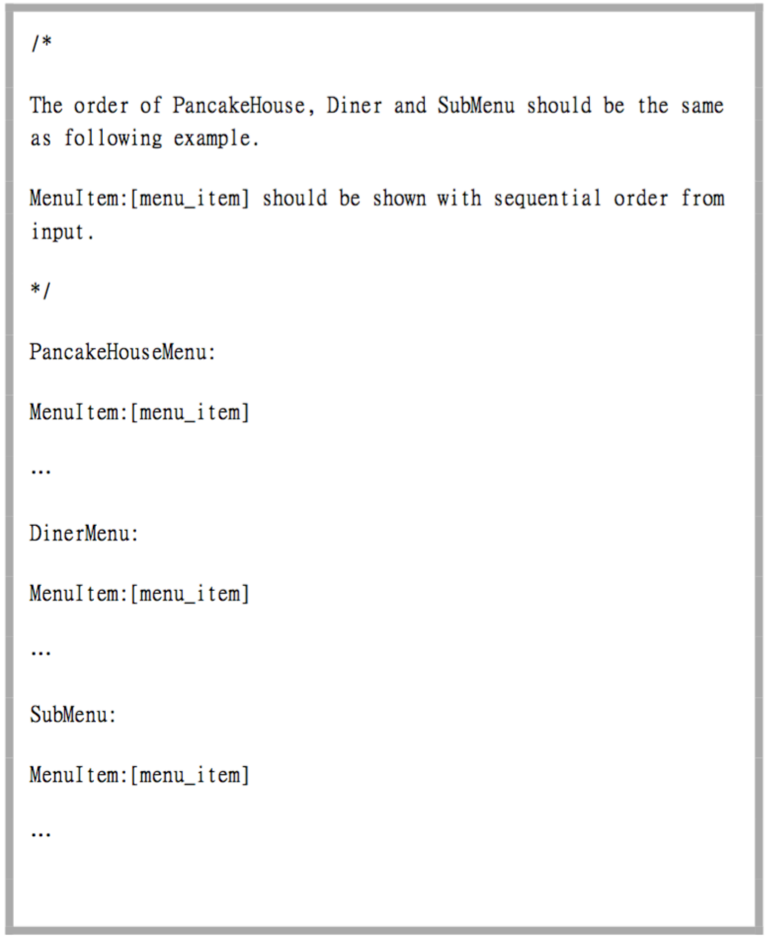
Input / Output

Input:



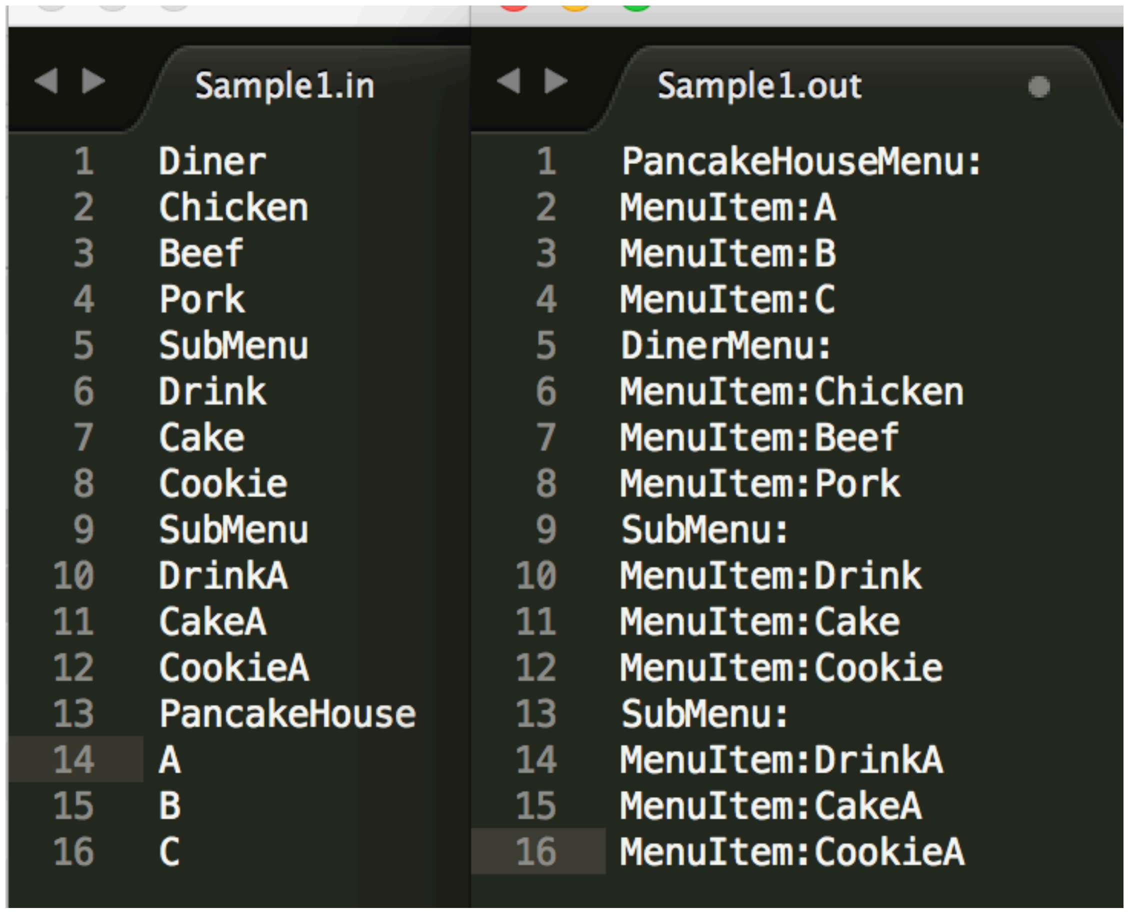
```
/*
The order of PancakeHouse and Diner could be different from
following example.
If SubMenu exists, it must follow Diner. It also means SubMenu
should not appear without Diner.
SubMenu could appear more than once or zero.
[menu_item] should be a string
*/


PancakeHouse
[menu_item]
…
Diner
[menu_item]
…
SubMenu
[menu_item]
…
```

Output:



```
/*
The order of PancakeHouse, Diner and SubMenu should be the same
as following example.
MenuItem:[menu_item] should be shown with sequential order from
input.
*/
PancakeHouseMenu:
MenuItem:[menu_item]
…
DinerMenu:
MenuItem:[menu_item]
…
SubMenu:
MenuItem:[menu_item]
…
```

Test case



Sample1.in

```
Diner
Chicken
Beef
Pork
SubMenu
Drink
Cake
Cookie
SubMenu
DrinkA
CakeA
CookieA
PancakeHouse
A
B
C
```

Sample1.out

```
PancakeHouseMenu:
MenuItem:A
MenuItem:B
MenuItem:C
DinerMenu:
MenuItem:Chicken
MenuItem:Beef
MenuItem:Pork
SubMenu:
MenuItem:Drink
MenuItem:Cake
MenuItem:Cookie
SubMenu:
MenuItem:DrinkA
MenuItem:CakeA
MenuItem:CookieA
```

BPEL Engine

Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering

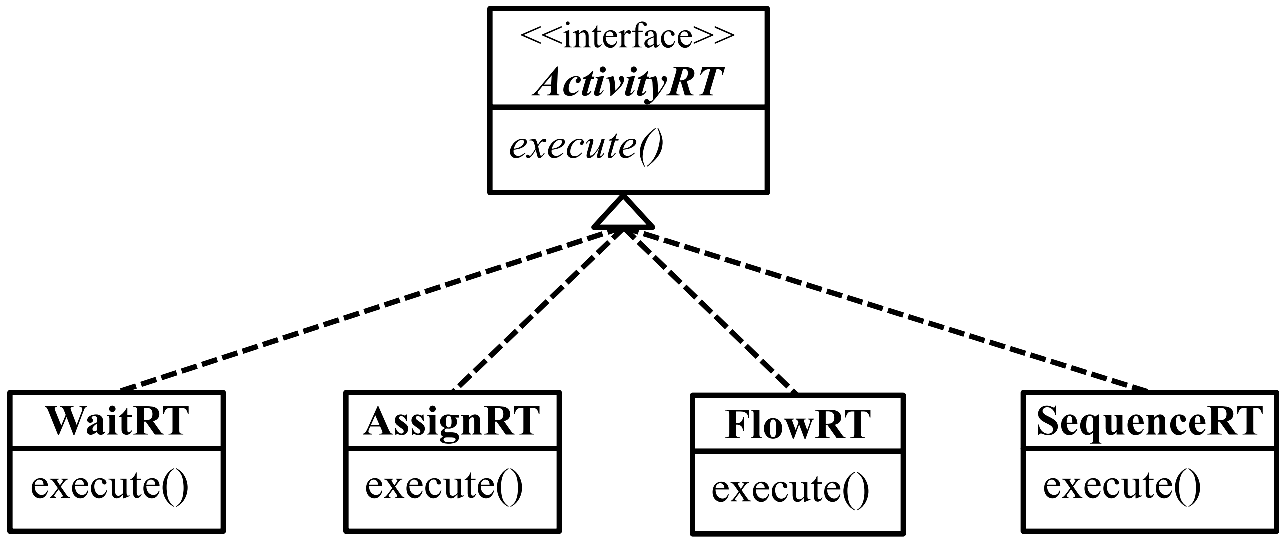
National Taiwan University

BPEL Engine

- A BPEL engine has several activities designed to perform the process logic, including WaitRT, AssignRT, Flow RT and SequenceRT.
- Among those activities, FlowRT and SequenceRT can contain multiple activities.

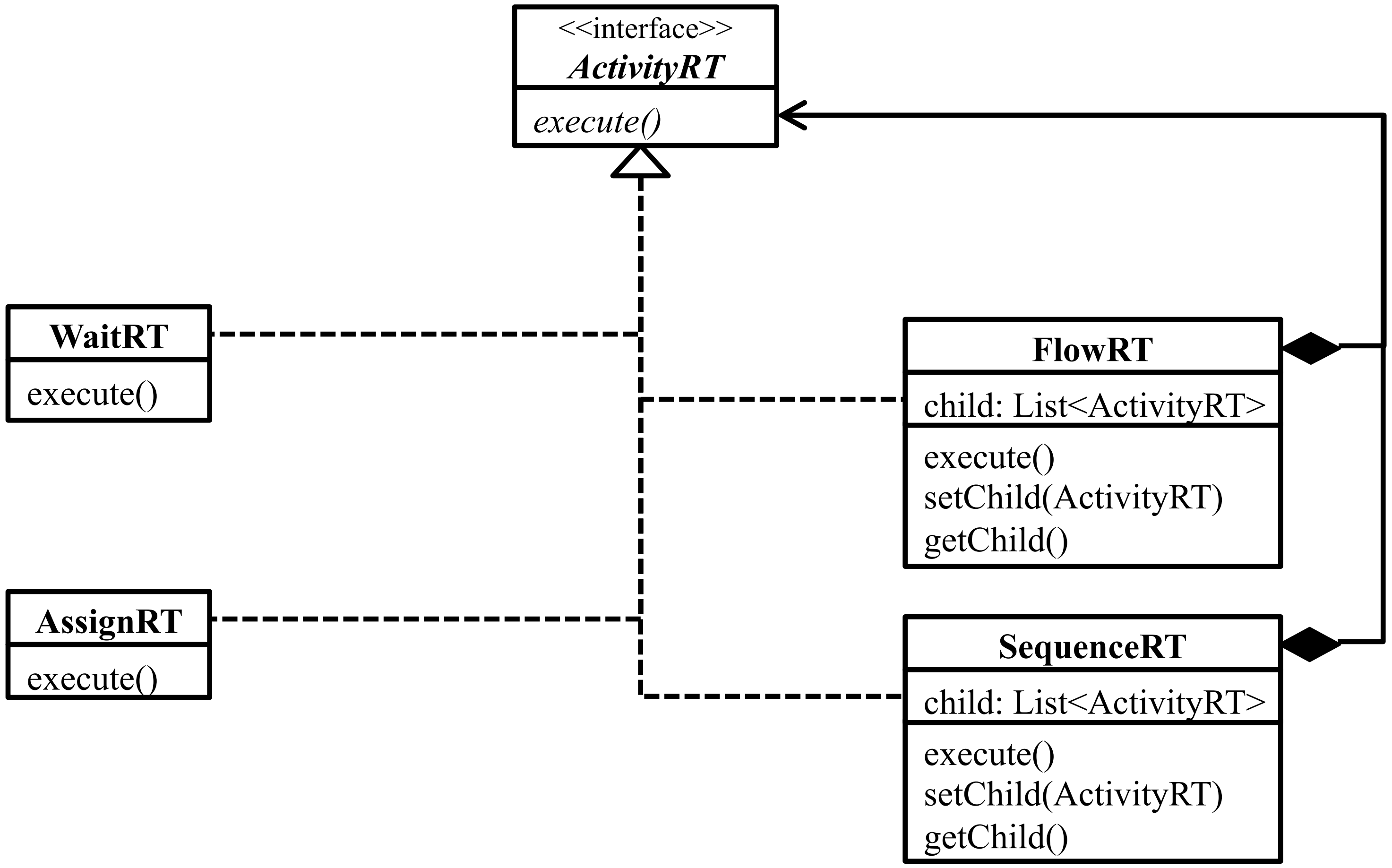
Requirements Statements$_1$

- A BPEL engine has several activities designed to perform the process logic, including WaitRT, AssignRT, Flow RT and SequenceRT.

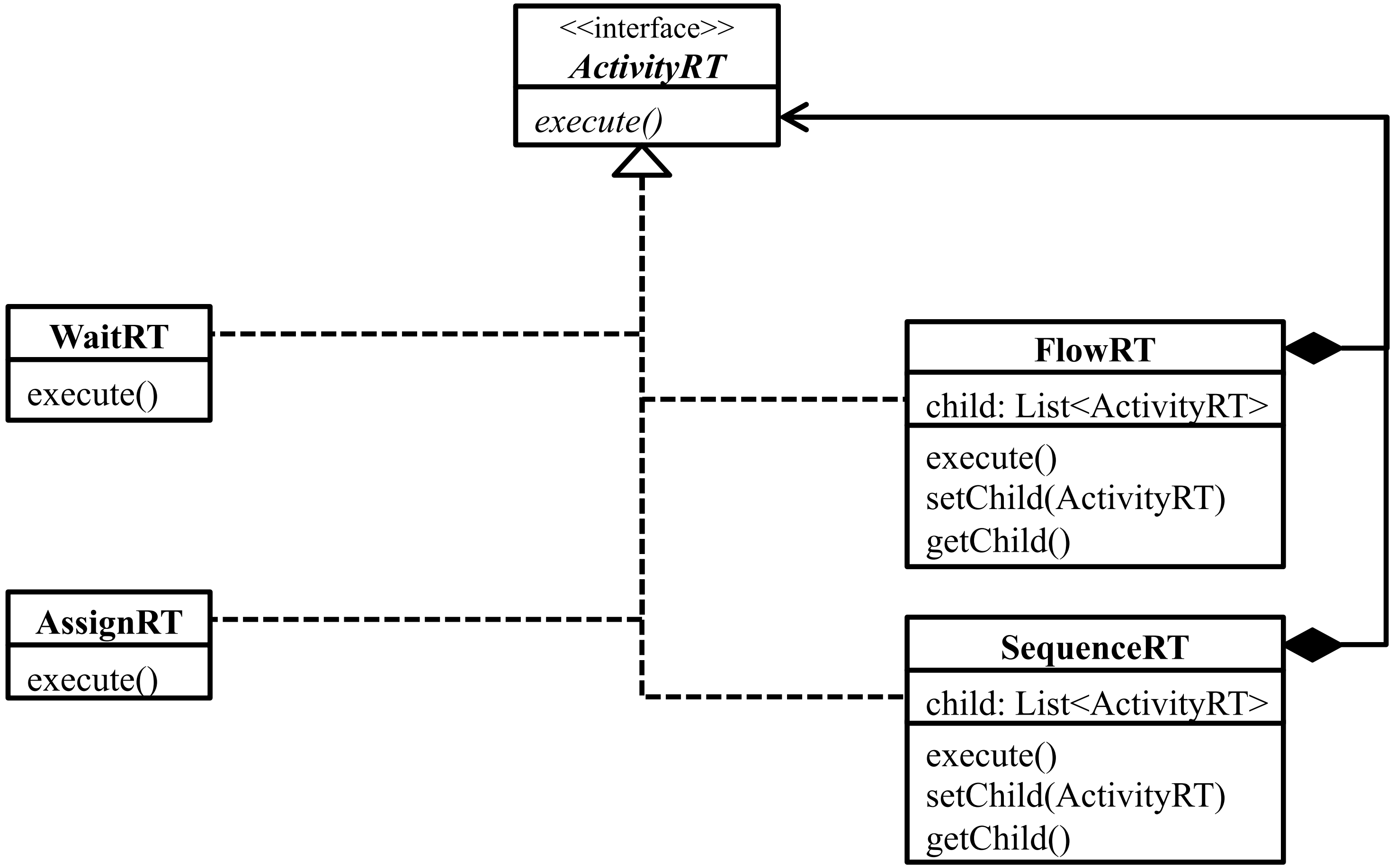


Requirements Statements$_2$

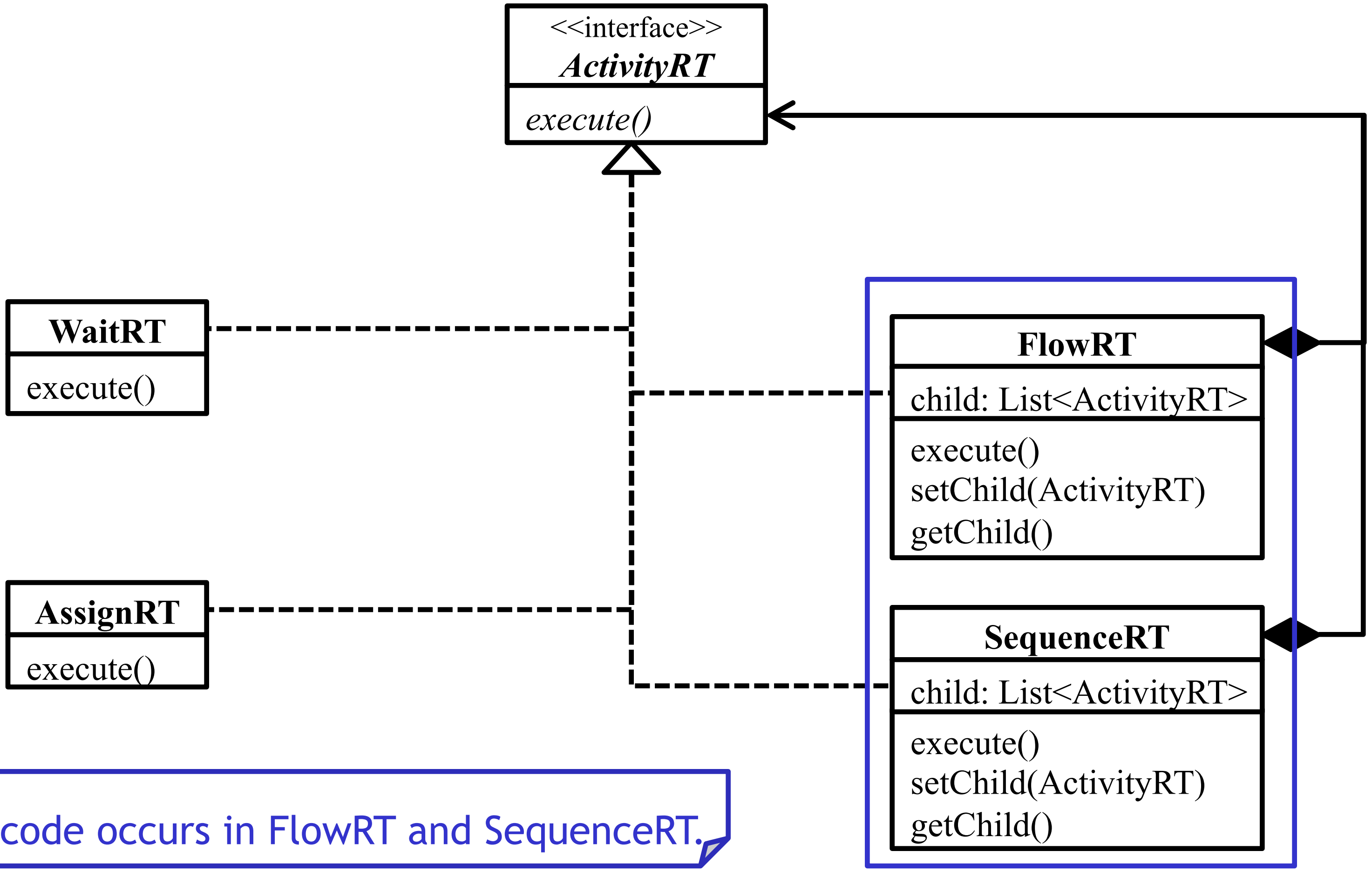
- Among those activities, FlowRT and SequenceRT can contain multiple activities.



Initial Design

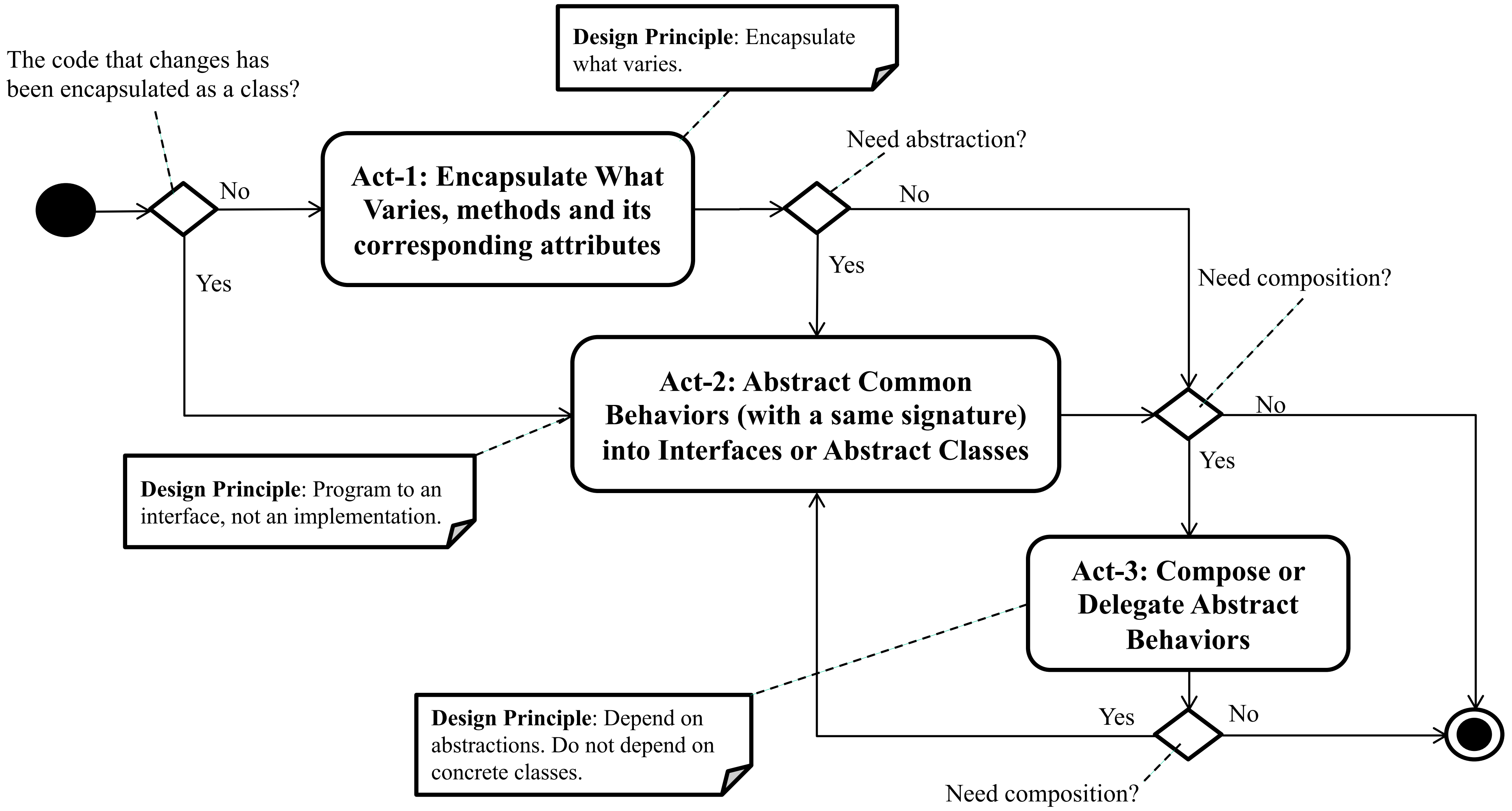


The Problem with the Initial Design



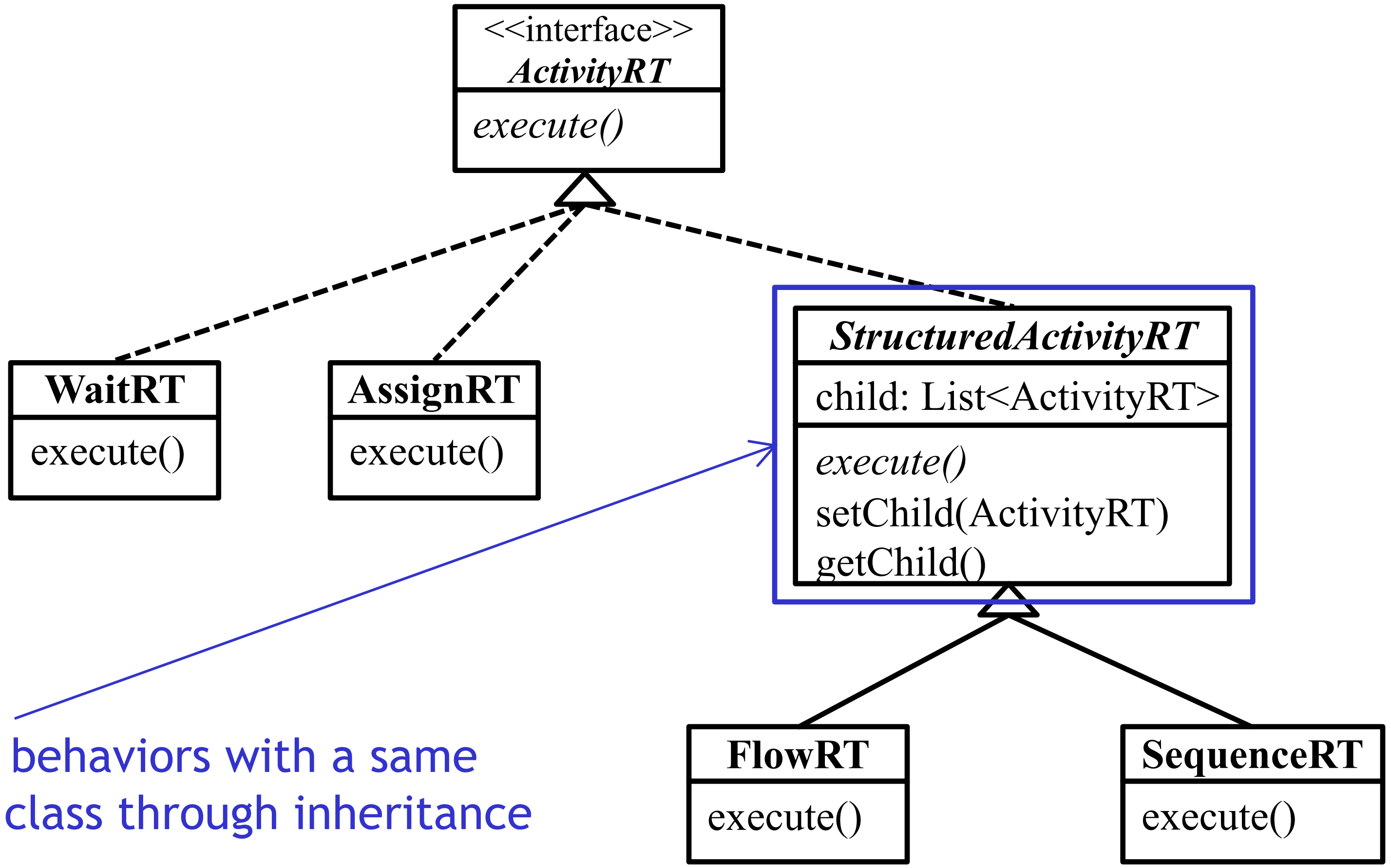
Problem: Duplicate code occurs in FlowRT and SequenceRT.

Design Process for Change



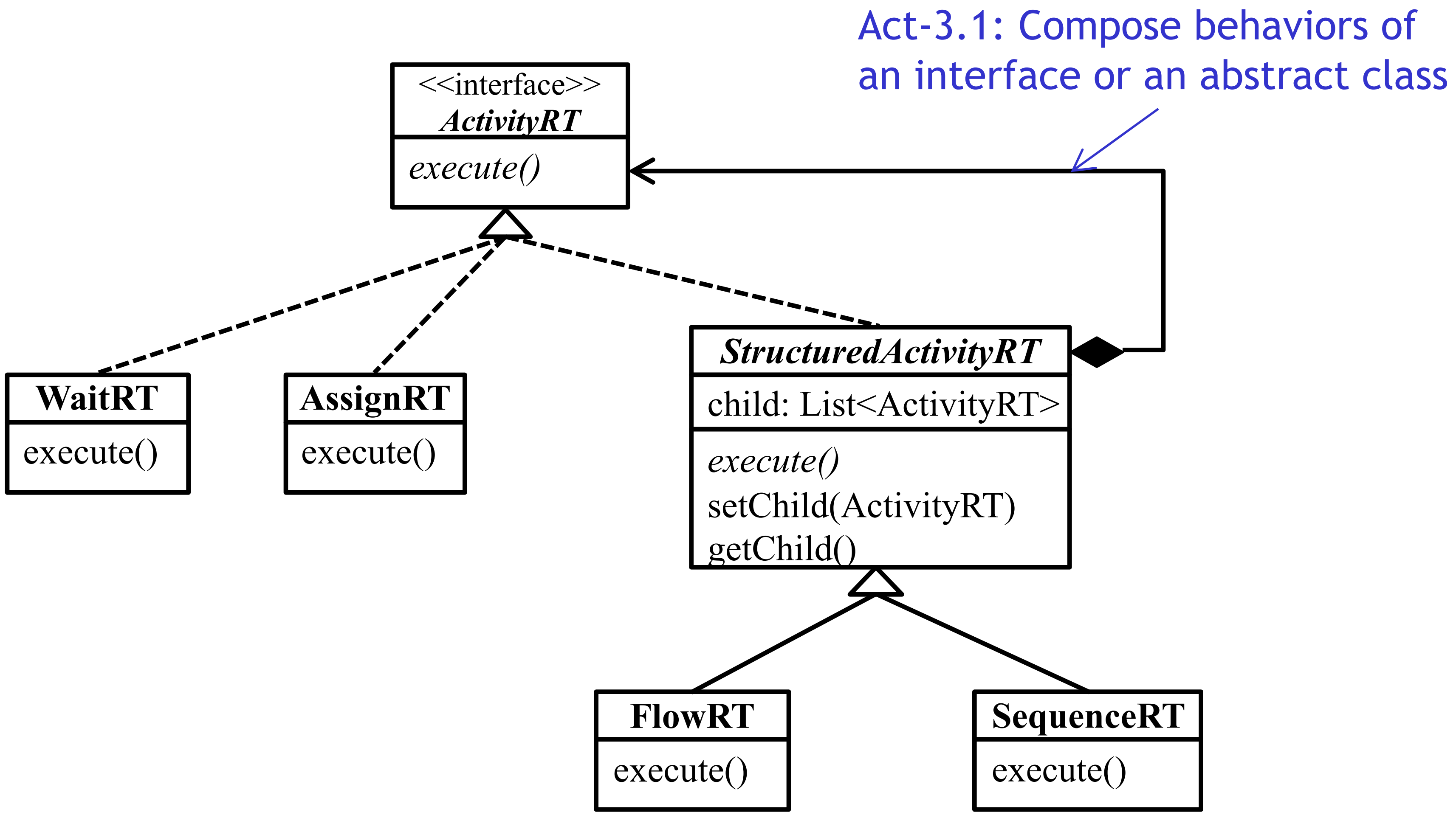
The code that changes has been encapsulated as a class?
Design Principle: Encapsulate what varies.
No
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

Act-2: Abstract Common Behaviors

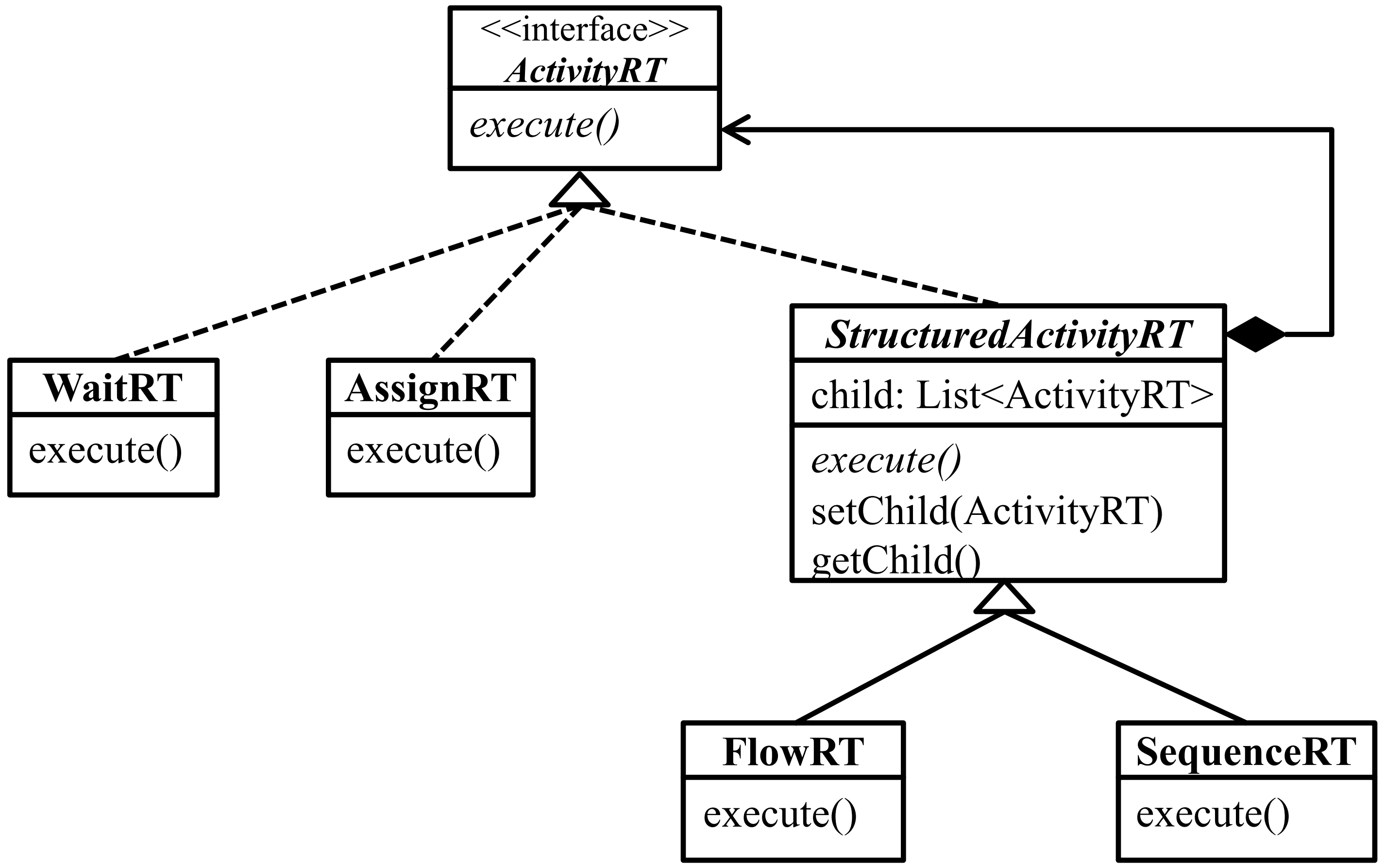


Act-2.1: Abstract common behaviors with a same signature into an abstract class through inheritance

Act-3: Compose Abstract Behaviors



Refactored Design after Design Process



<<interface>>
ActivityRT
execute()
WaitRT
execute()
AssignRT
execute()
StructuredActivityRT
child: List<ActivityRT>
execute()
setChild(ActivityRT)
getChild()
FlowRT
execute()
SequenceRT
execute()