

Composite Pattern

Prof. Jonathan Lee (李允中)

Department of CSIE

National Taiwan University



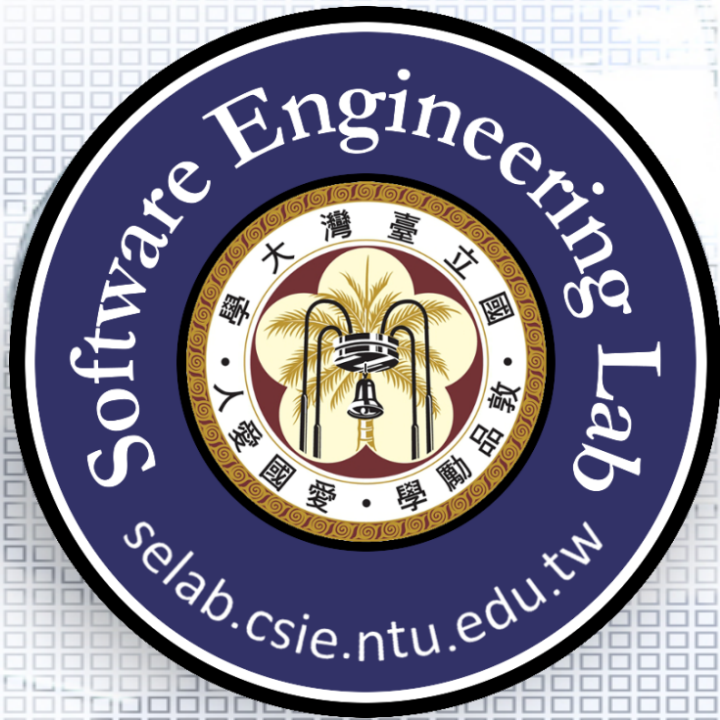
Design Aspect of Composite

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



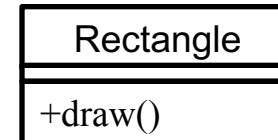
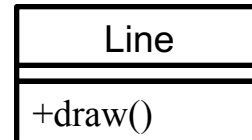
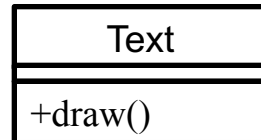
Requirements Statement

- ❑ 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.



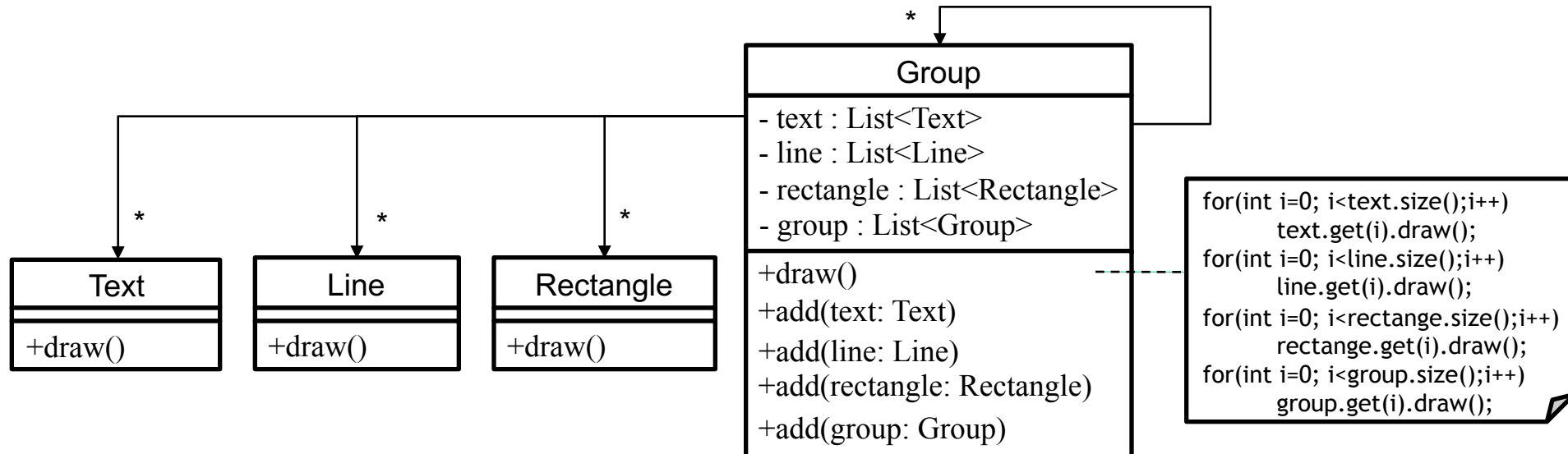
Requirements Statements₁

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

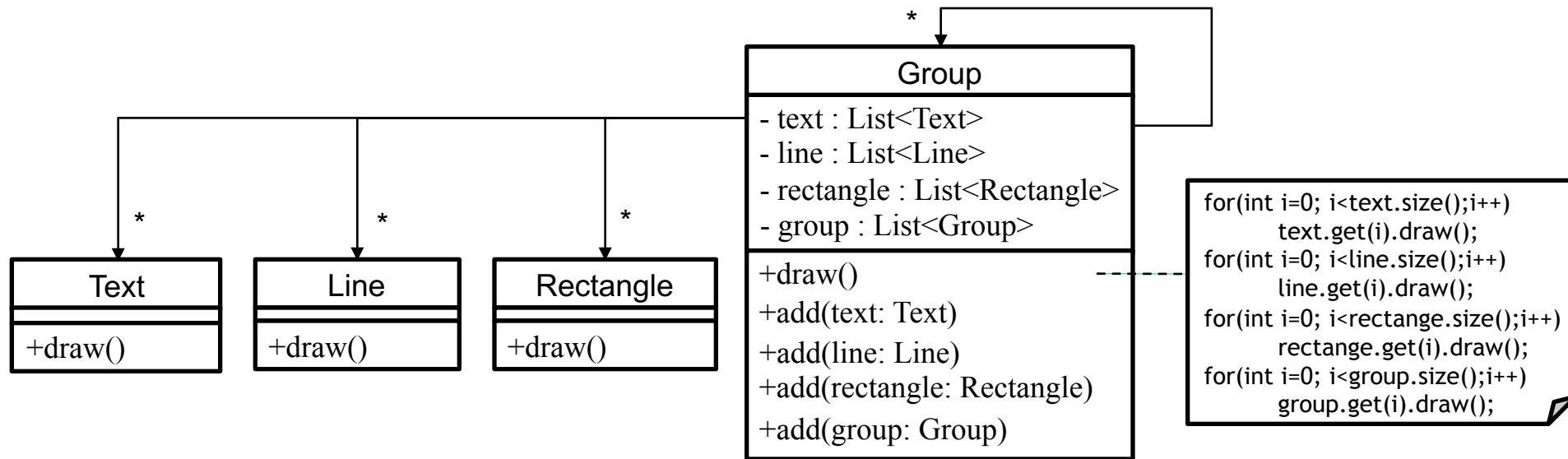


Requirements Statements₂

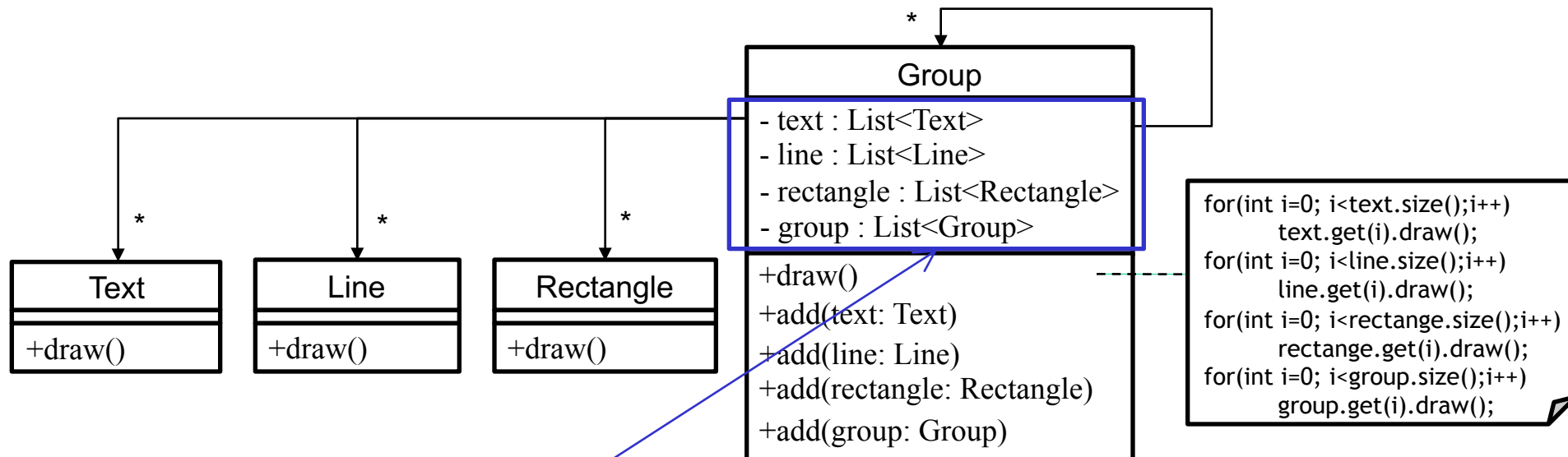
- ❑ The user can group basic components to form larger components, which in turn can be grouped to form still larger components.



Initial Design

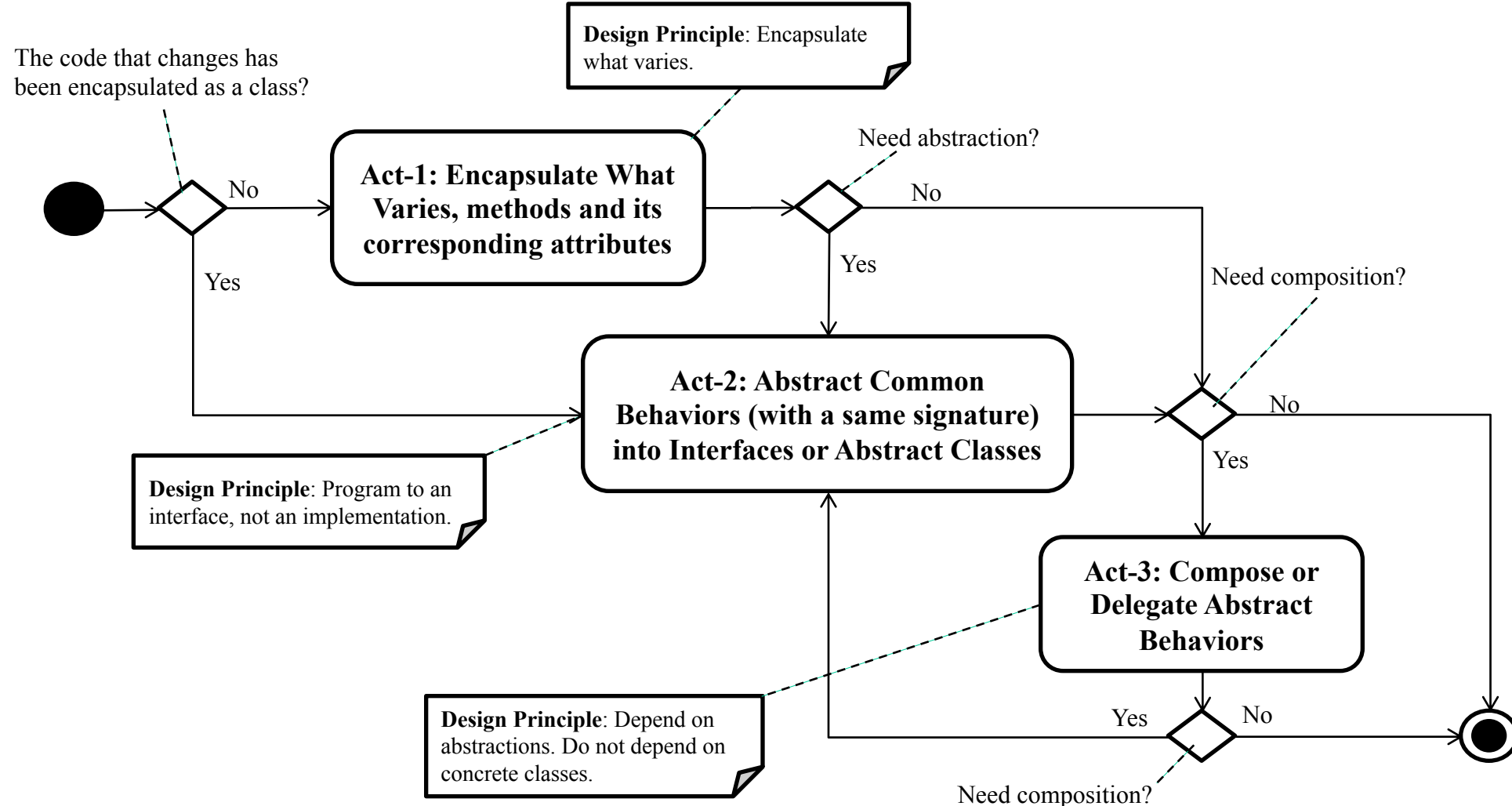


Problems with Initial Design



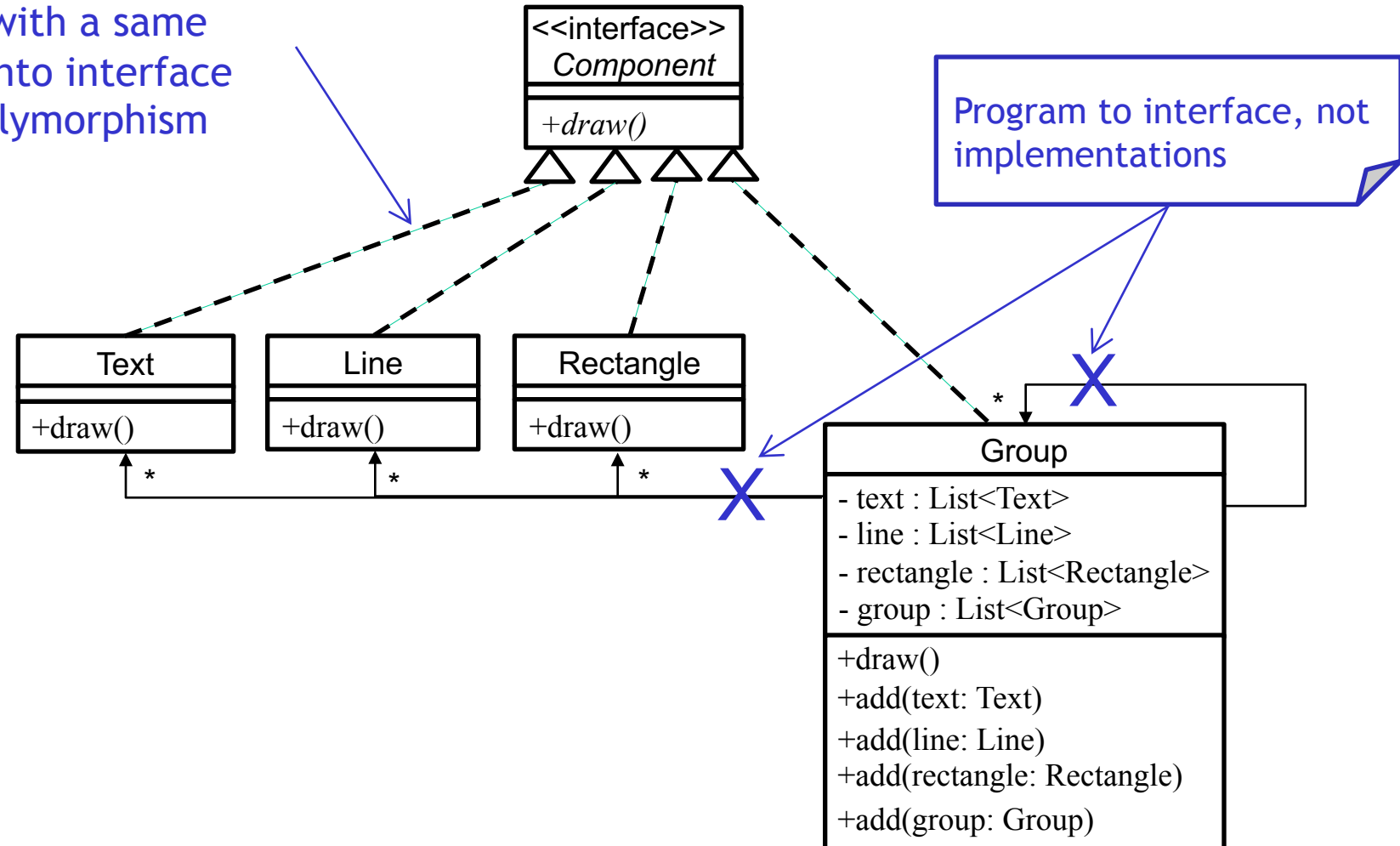
Problem : If a new requirement is to add a new basic component such as Triangle, then we need to modify Group to meet the new requirement.

Design Process for Change



Act-2: Abstract Common Behaviors

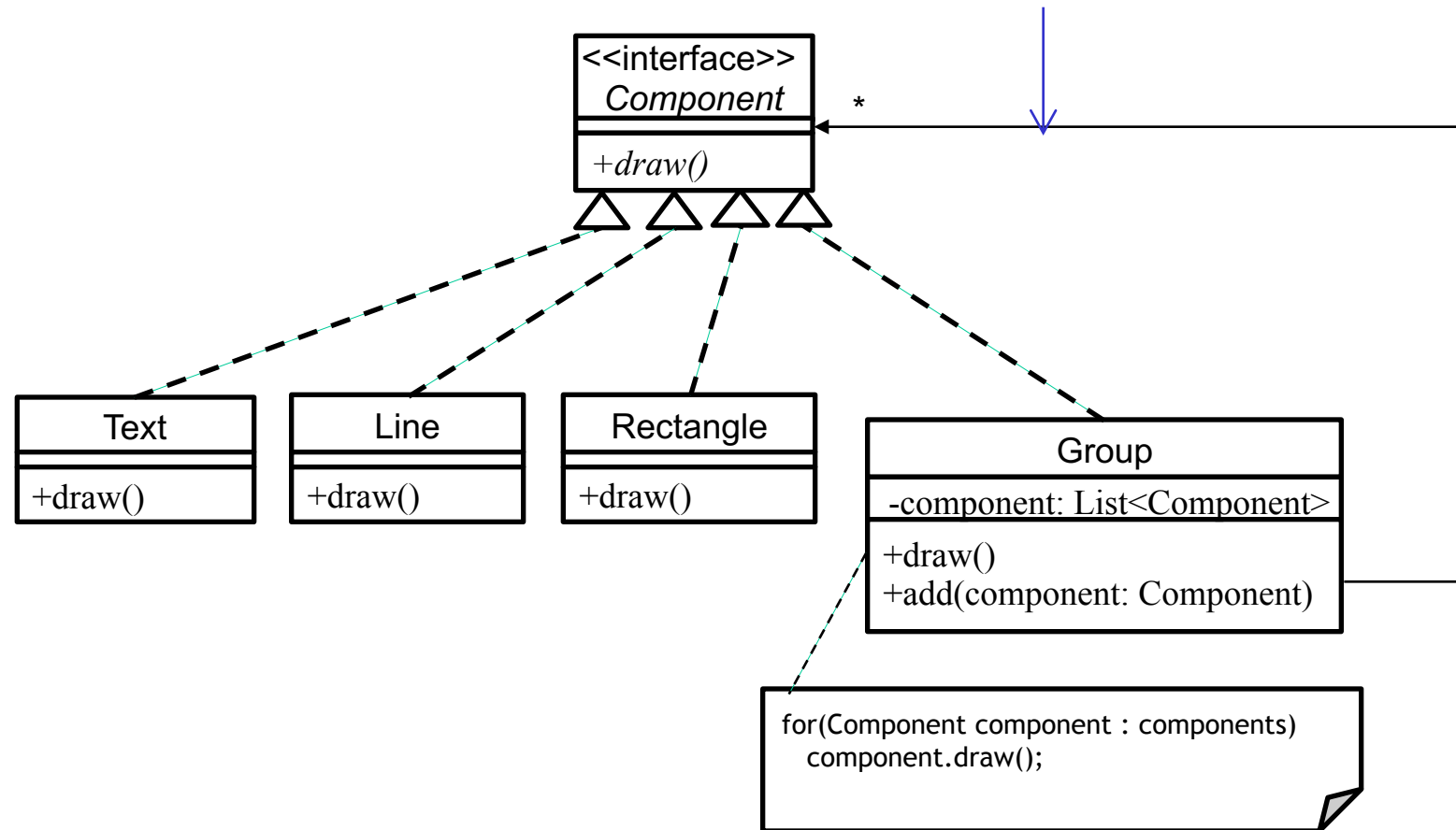
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism





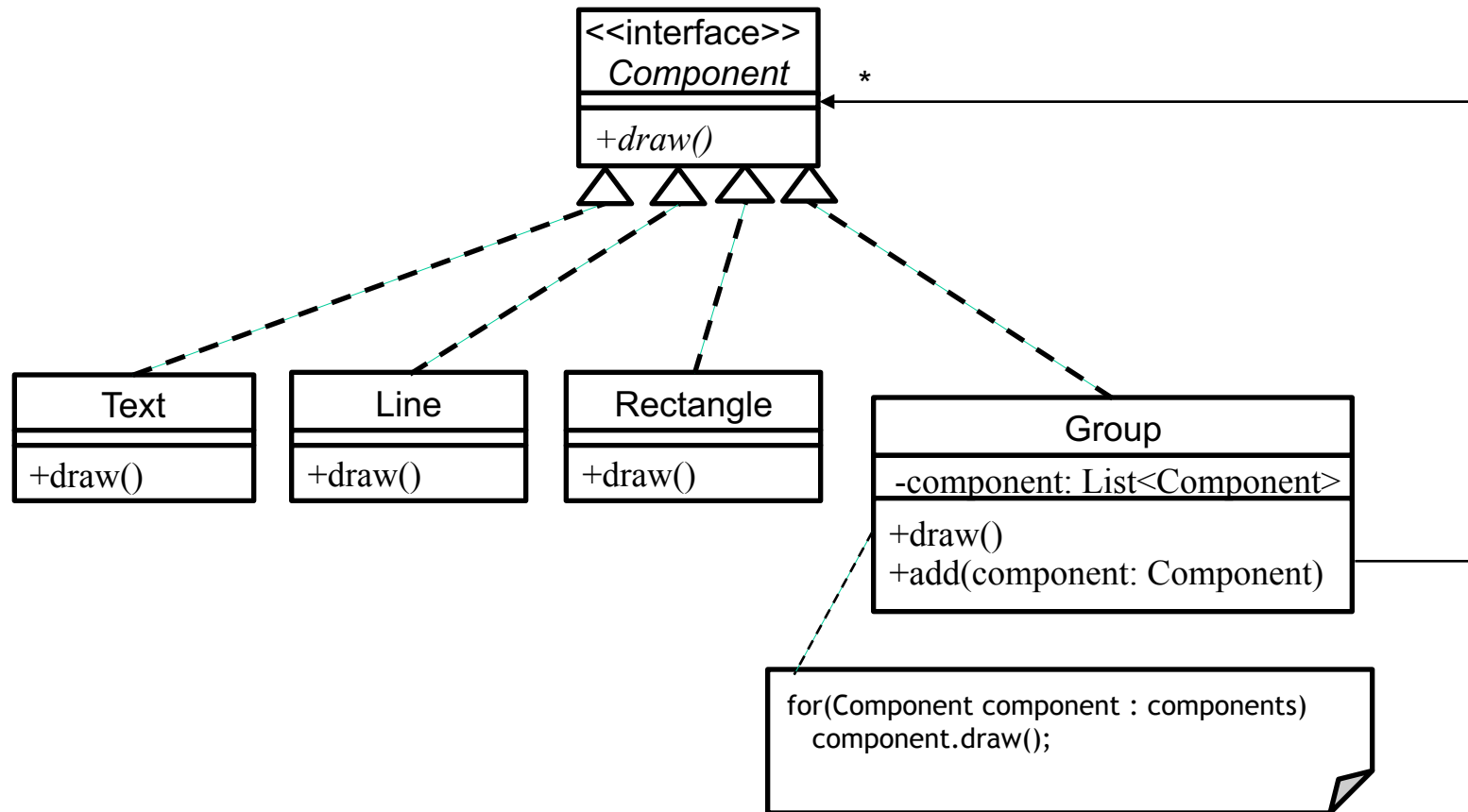
Act-3: Compose Abstract Behaviors

Act-3.1: Compose behaviors of an interface or an abstract class





Refactored Design after Design Process





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.

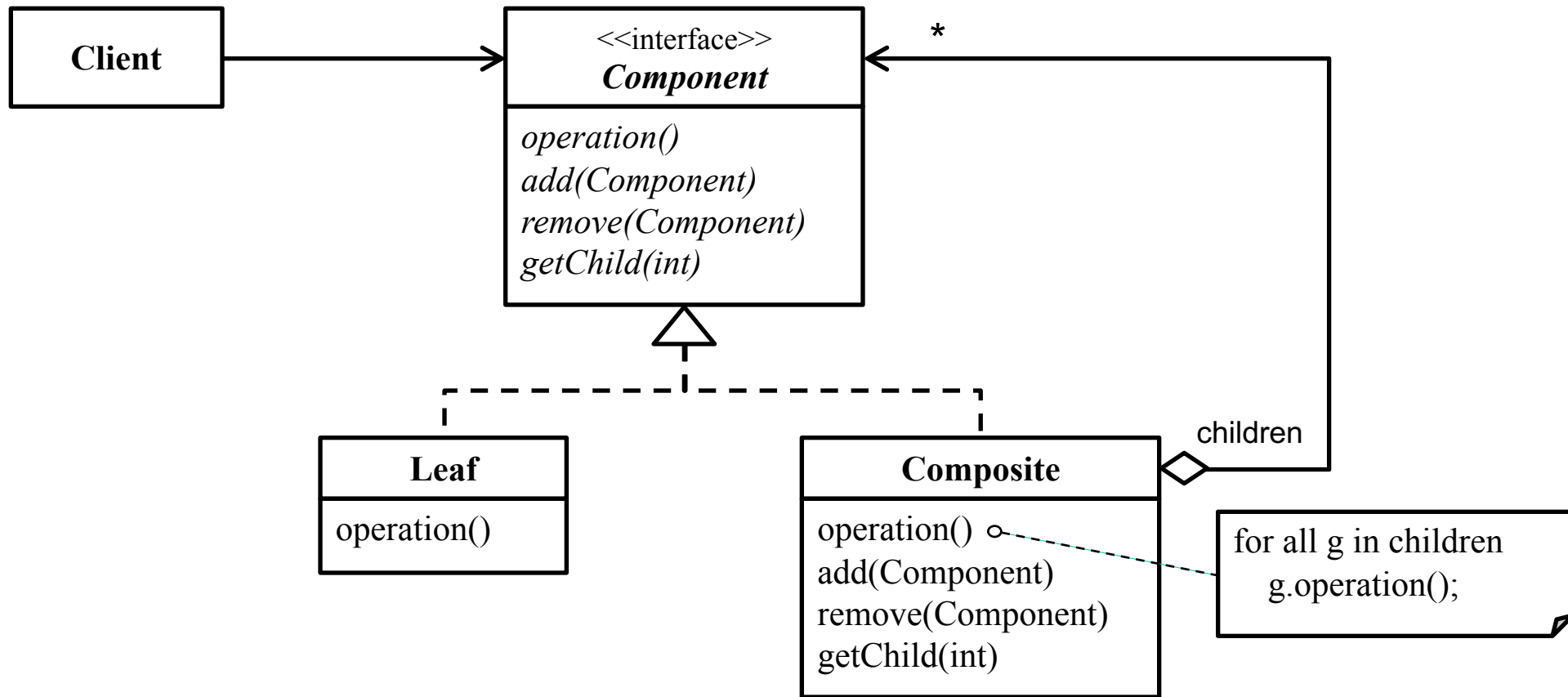


Intent

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

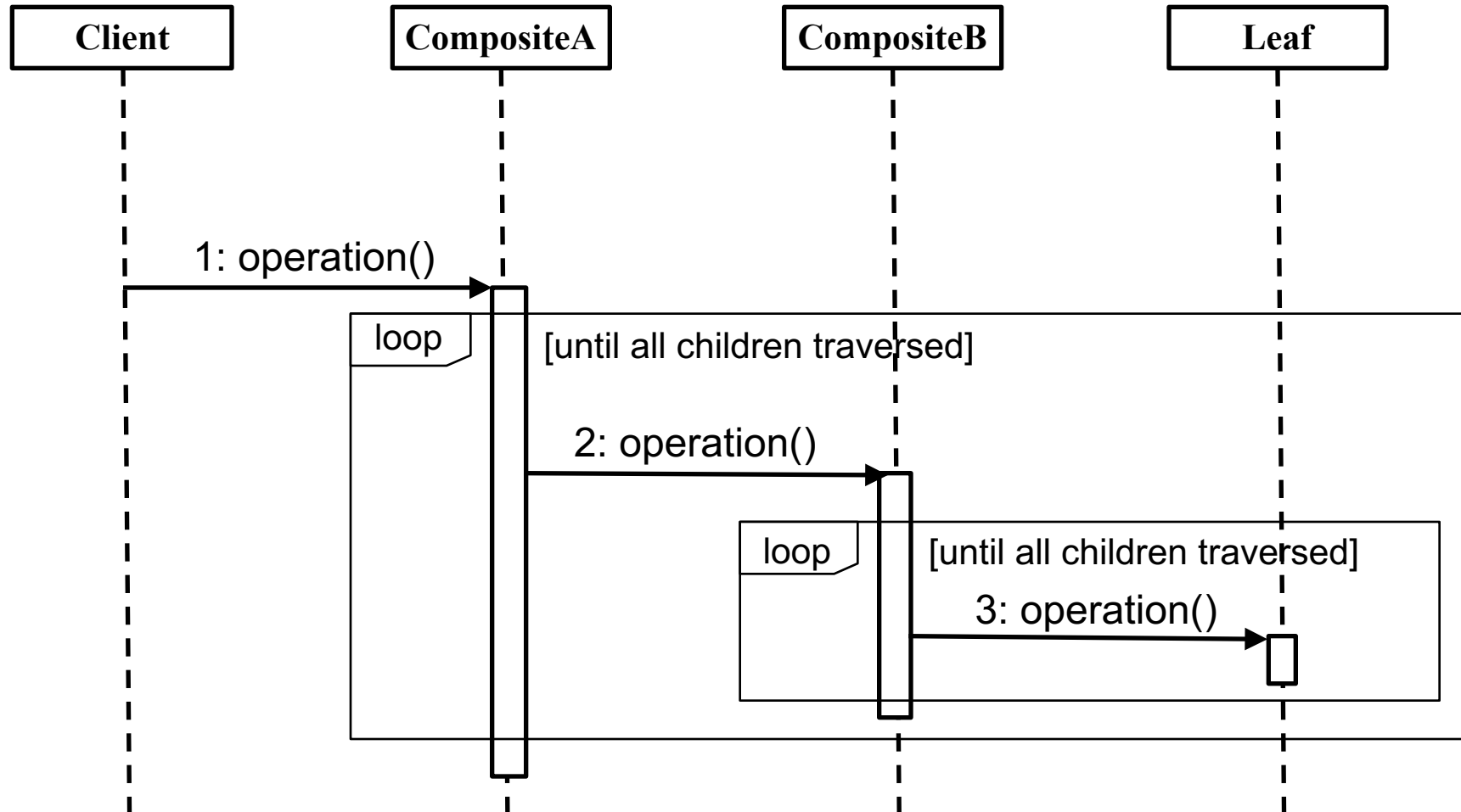


Composite Pattern Structure₁





Composite Pattern Structure₂





Composite Pattern Structure₃

	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

```
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	Sample1.out
1 <?xml version="1.0"?>	1 Line
2 <Question>	2 Text
3 <Line/>	3 Rectangle
4 <Text/>	4 Group: {}
5 <Rectangle/>	
6 <Group/>	
7 </Question>	



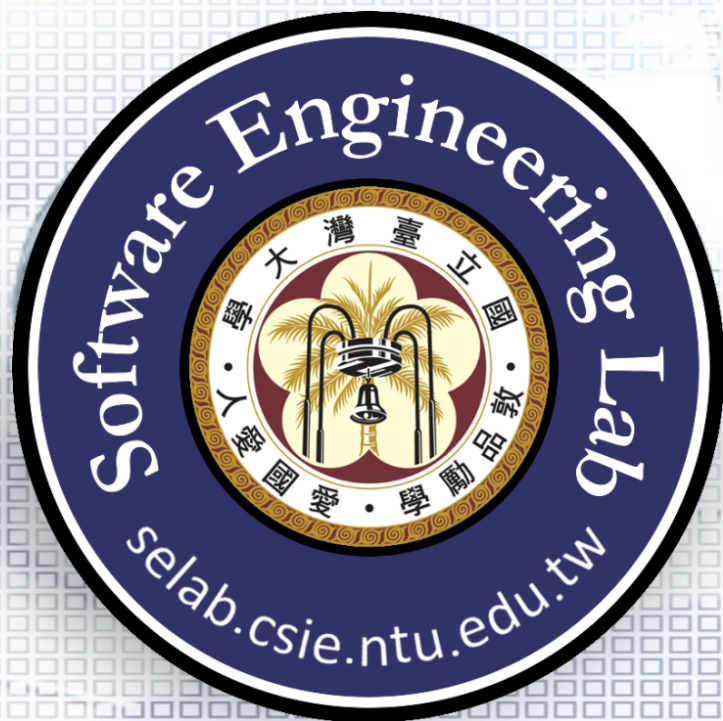
Test case2

```
Sample2.in                                     Sample2.out                                     Sample
1  <?xml version="1.0"?>                      1  Group:{Line Text Rectangle }
2  <Question>                                    2
3    <Group>
4      <Line/>
5      <Text/>
6      <Rectangle/>
7    </Group>
8  </Question>
```



Test case3

```
Sample3.in                                     Sample3.out
1 <?xml version="1.0"?>                        1 Group:{Line Text Rectangle Group:{} Group:{Line Text Line Text Rectangle Rectangle Group:{} Rectangle } }
2 <Question>                                     2 Rectangle
3   <Group>                                       3 Group:{}
4     <Line/>                                    4 Group:{Line Text Line Text Rectangle Group:{} Rectangle } |
5     <Text/>
6     <Rectangle/>
7   </Group>
8   <Group>
9     <Line/>
10    <Text/>
11    <Line/>
12    <Text/>
13    <Rectangle/>
14    <Rectangle/>
15  </Group>
16  <Rectangle/>
17 </Group>
18 </Question>
19 <Rectangle/>
20 <Group/>
21 <Group>
22   <Line/>
23   <Text/>
24   <Line/>
25   <Text/>
26   <Rectangle/>
27   <Group/>
28   <Rectangle/>
29 </Group>
30 </Question>
```



Extended Merge of Two Menus (Composite)

Prof. Jonathan Lee (李允中)

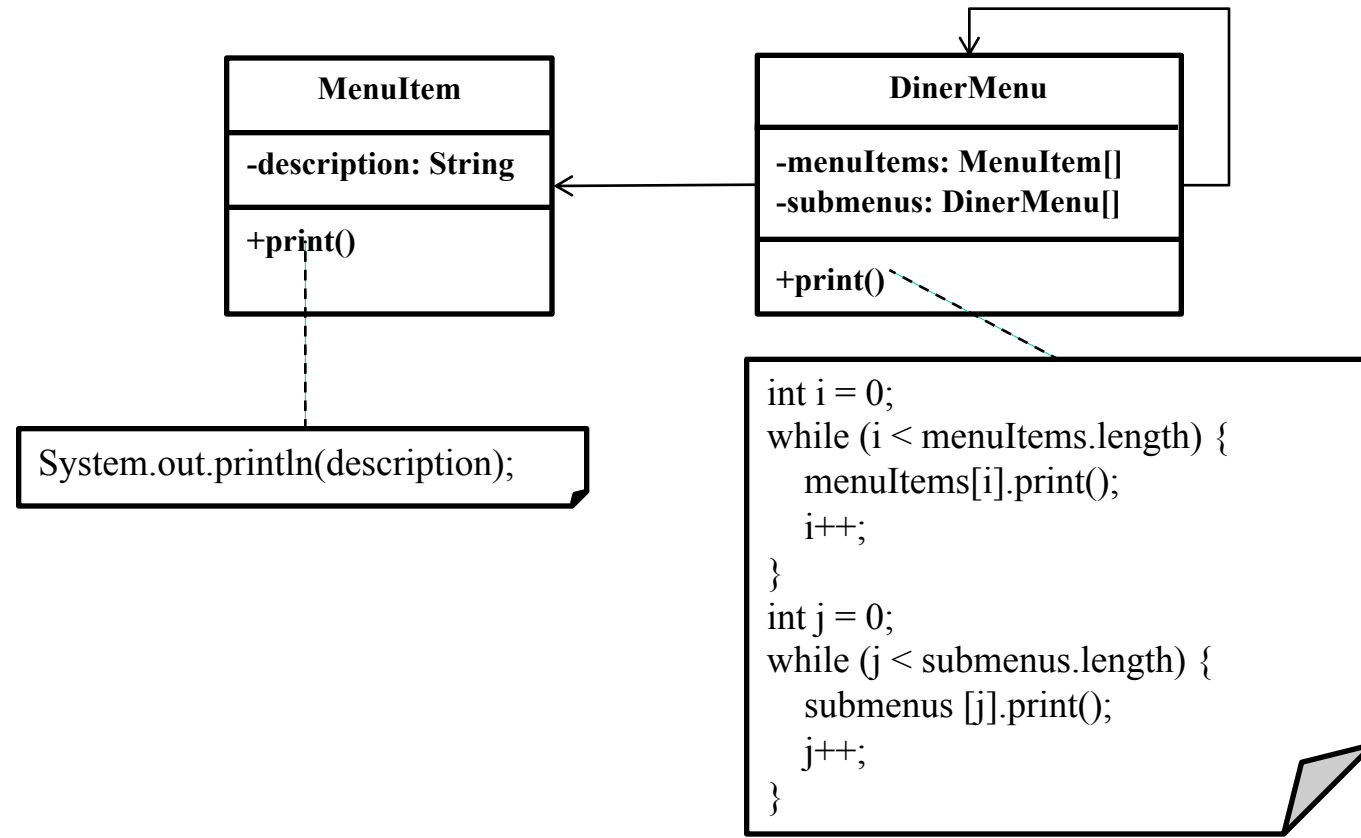
Department of Computer Science and
Information Engineering
National Taiwan University



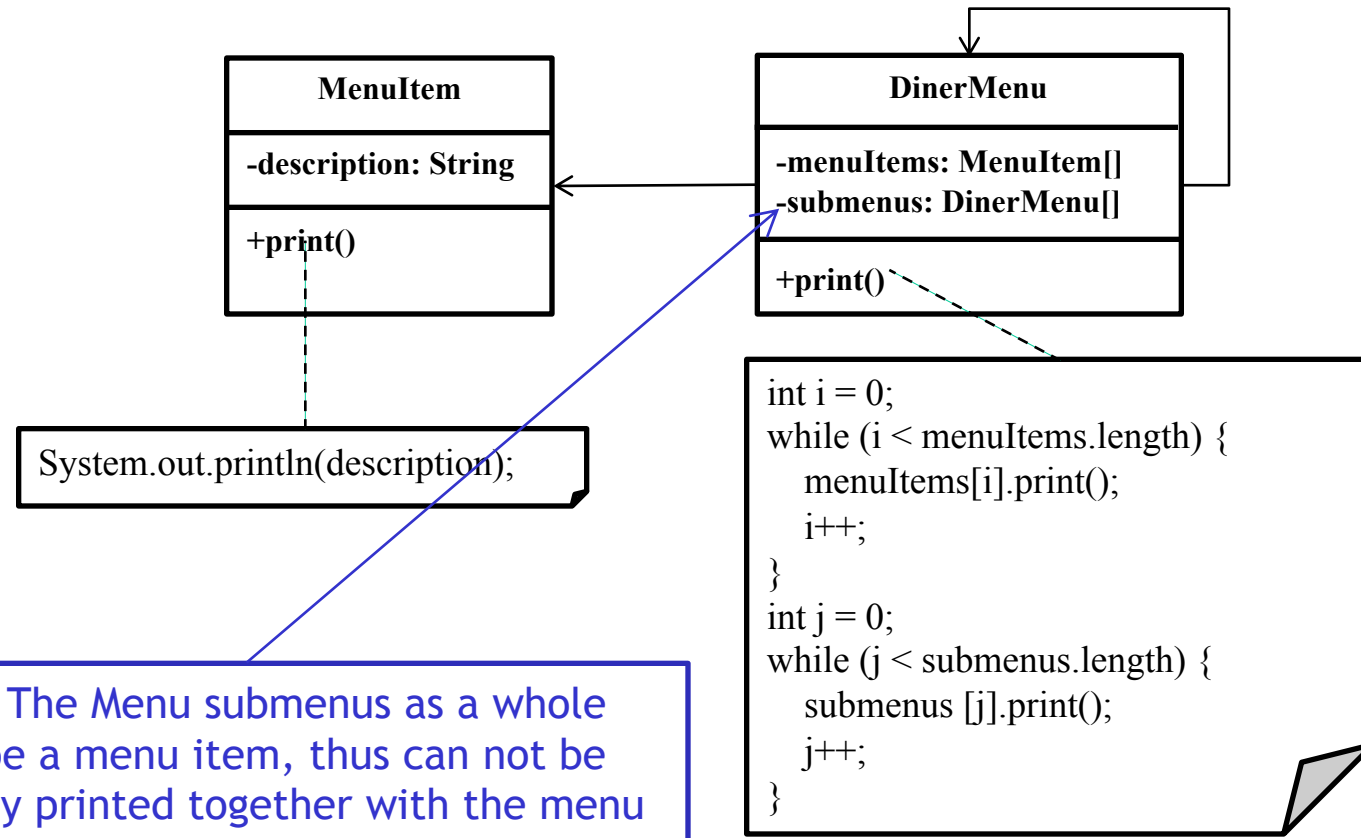
Requirements Statement

- ❑ 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

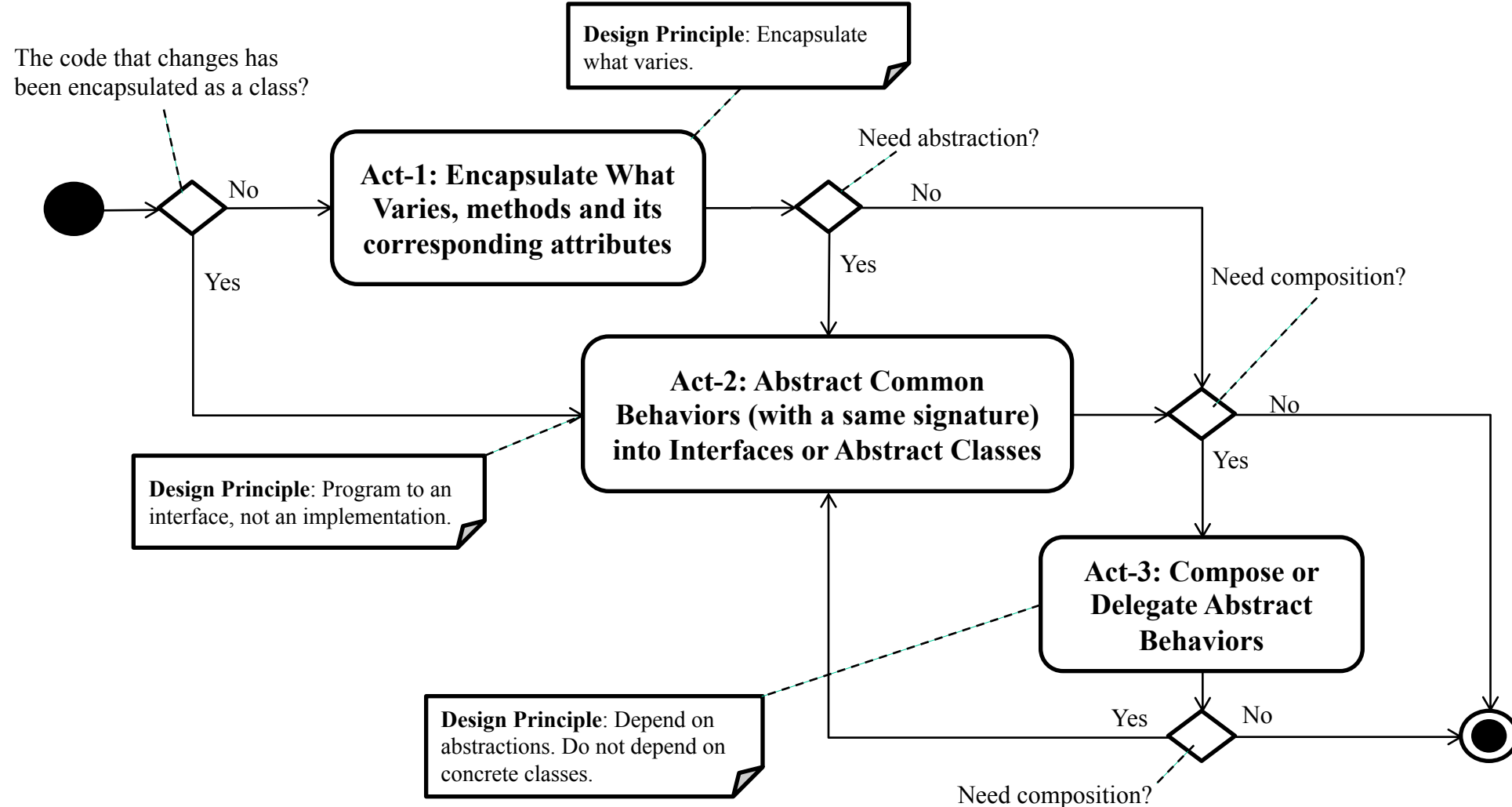


Problems with Initial Design

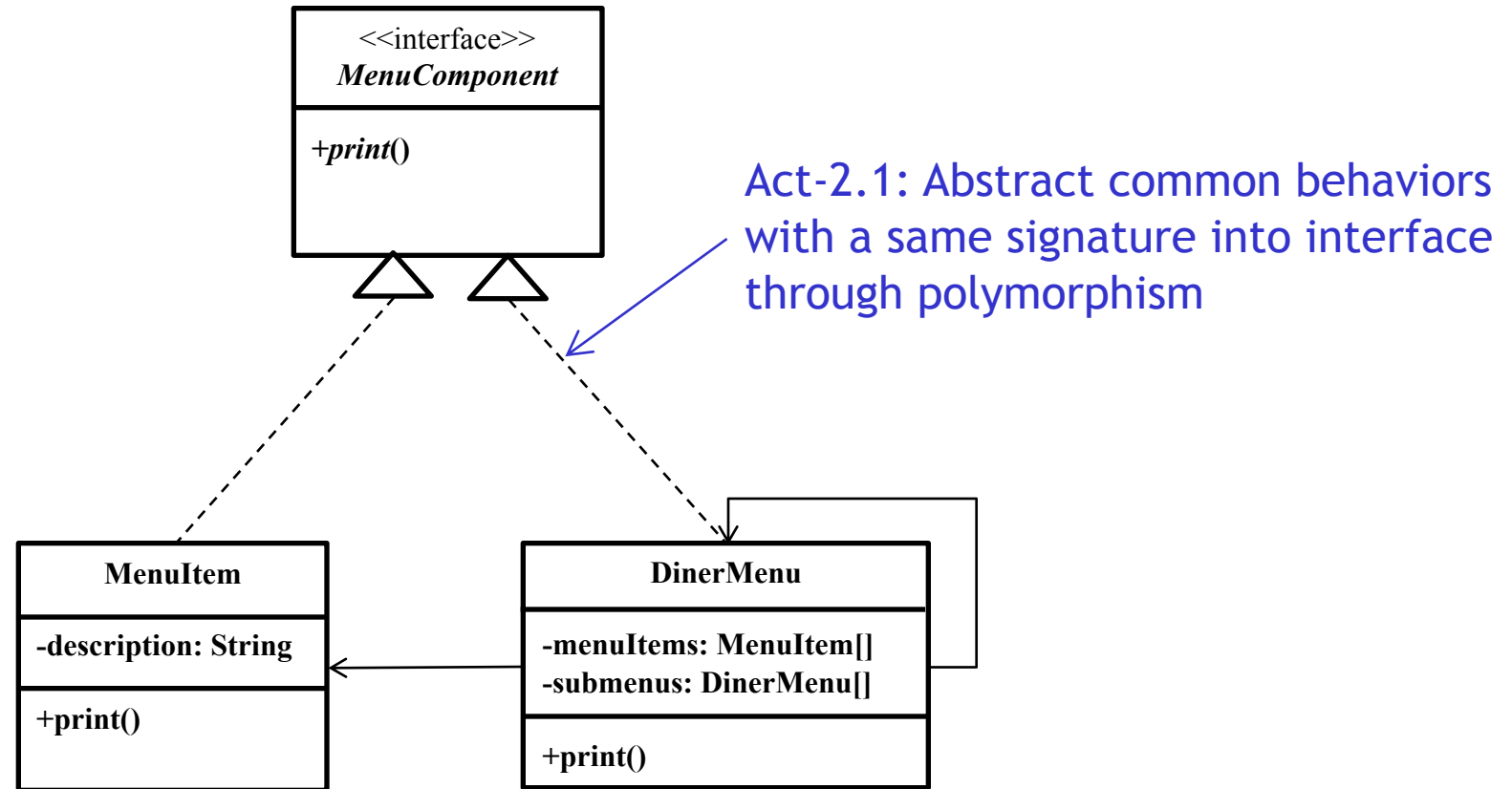


Problem: The Menu submenus as a whole can not be a menu item, thus can not be iteratively printed together with the menu items.

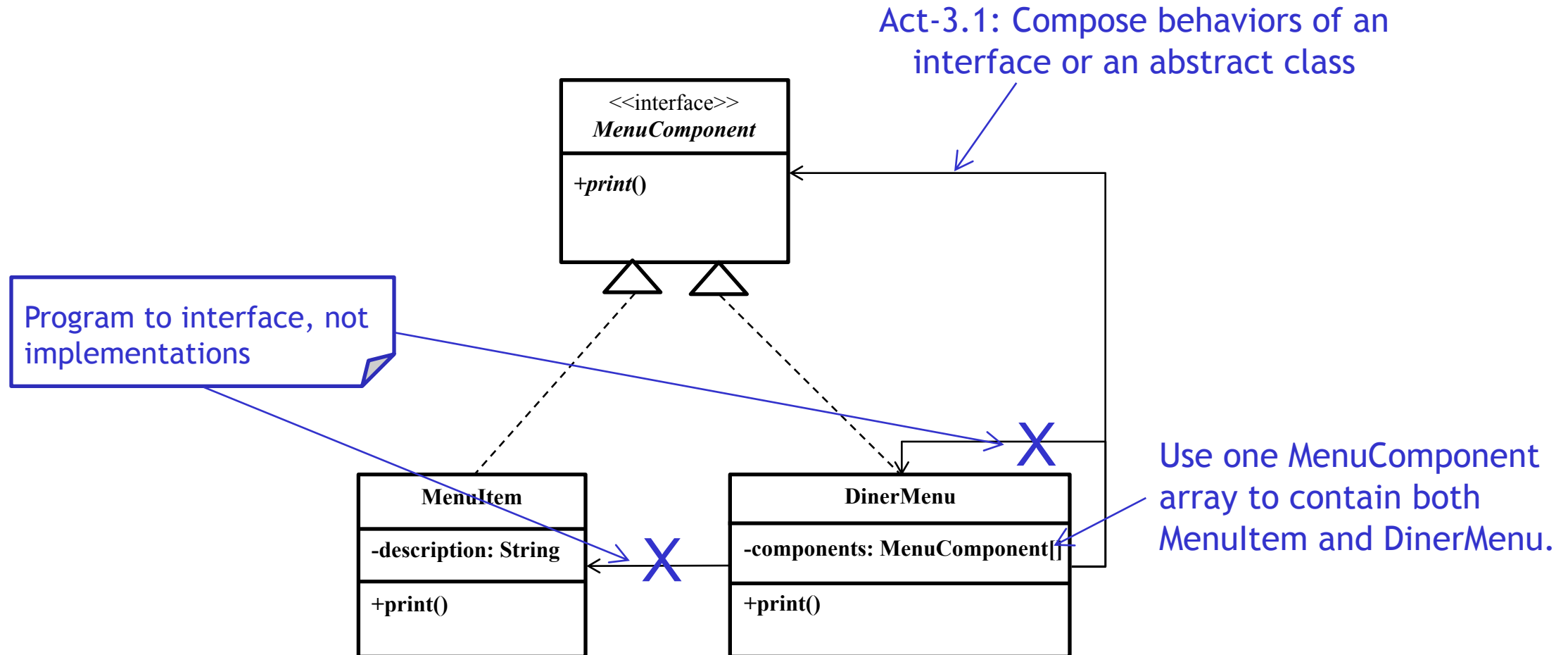
Design Process for Change



Act-2: Abstract Common Behaviors

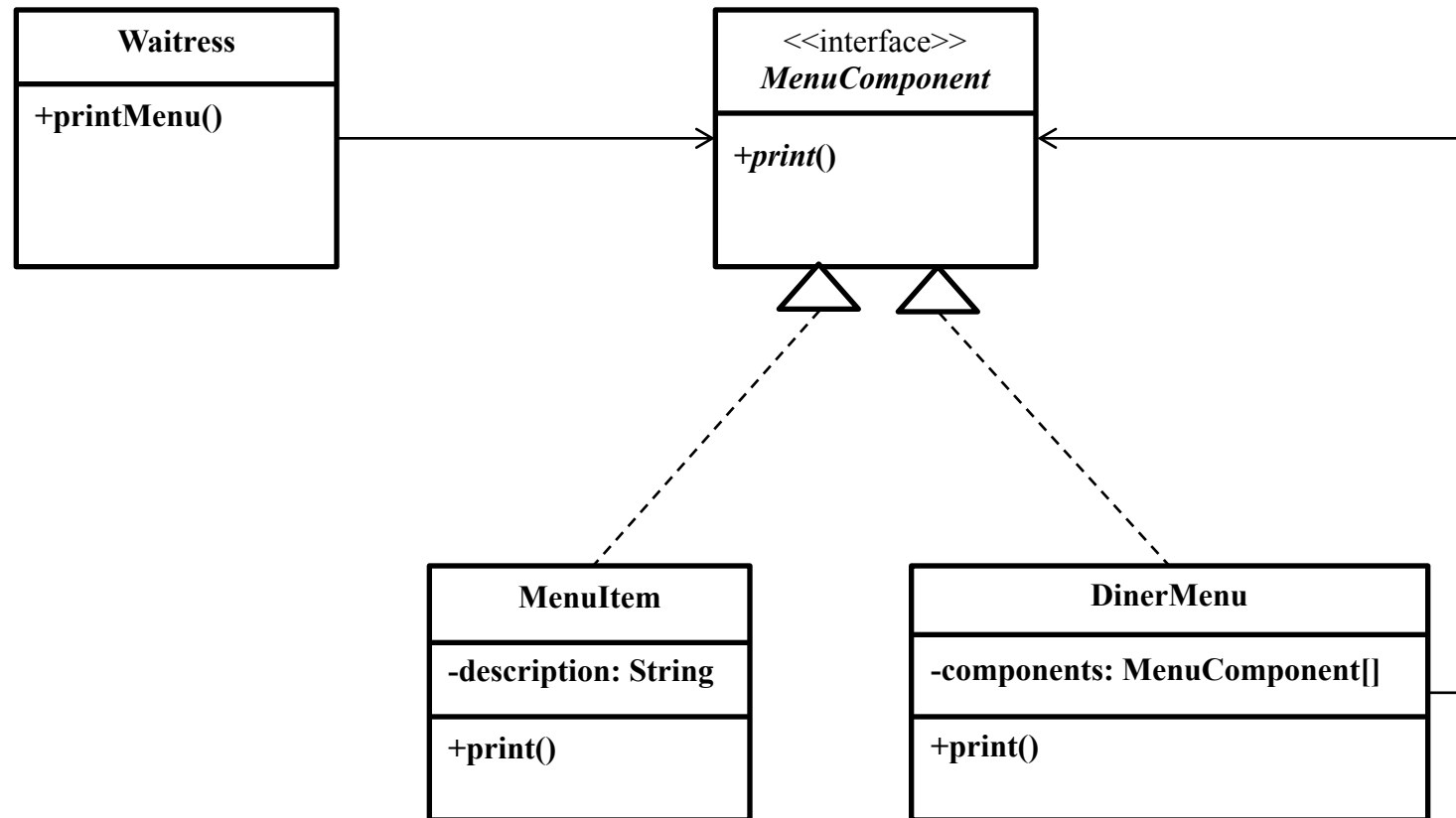


Act-3: Compose Abstract Behaviors



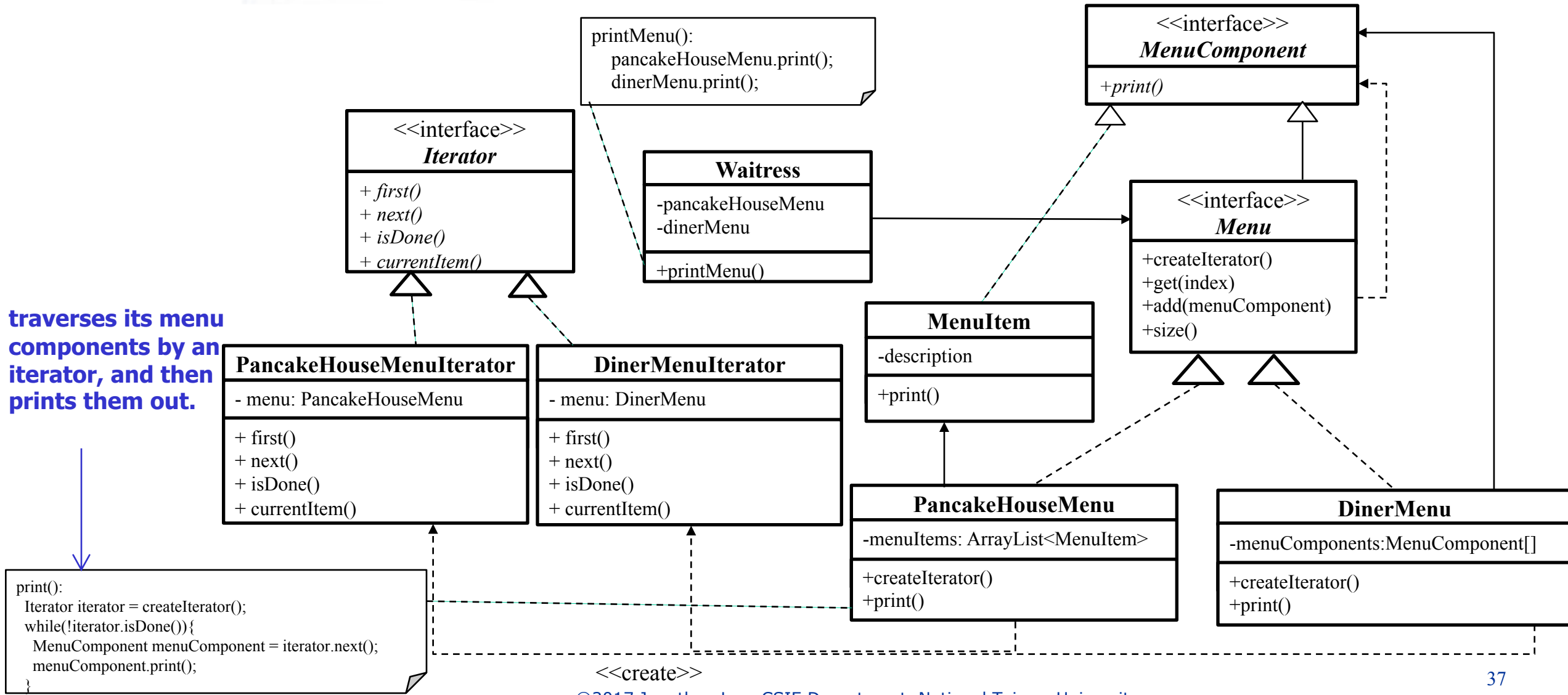


Refactored Design after Design Process





Print Menus in Composite with Iterator





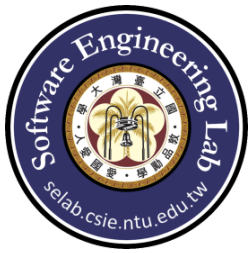
Waitress

```
public class Waitress {  
    private PancakeHouseMenu pancakeHouseMenu;  
    private DinerMenu 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

```
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

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



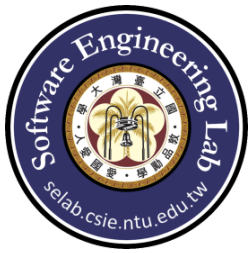
Input / Output

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]  
...
```

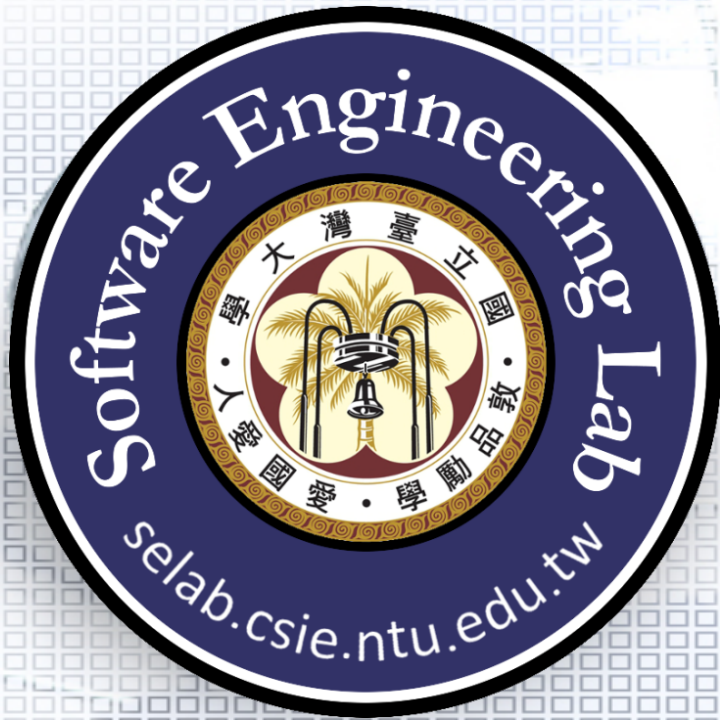
Output:

```
/*  
  
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	Sample1.out
1 Diner	1 PancakeHouseMenu:
2 Chicken	2 MenuItem:A
3 Beef	3 MenuItem:B
4 Pork	4 MenuItem:C
5 SubMenu	5 DinerMenu:
6 Drink	6 MenuItem:Chicken
7 Cake	7 MenuItem:Beef
8 Cookie	8 MenuItem:Pork
9 SubMenu	9 SubMenu:
10 DrinkA	10 MenuItem:Drink
11 CakeA	11 MenuItem:Cake
12 CookieA	12 MenuItem:Cookie
13 PancakeHouse	13 SubMenu:
14 A	14 MenuItem:DrinkA
15 B	15 MenuItem:CakeA
16 C	16 MenuItem:CookieA



BPEL Engine

Prof. Jonathan Lee (李允中)

Department of Computer Science and
Information Engineering
National Taiwan University



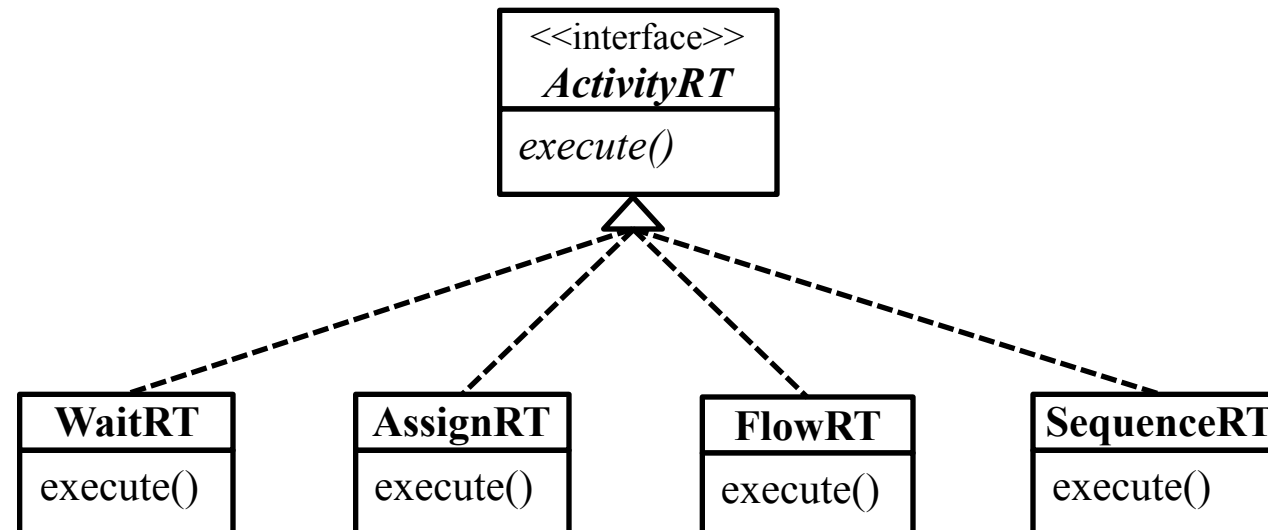
BPEL Engine

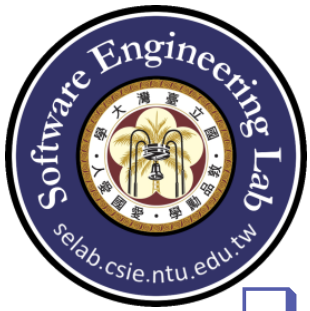
- ☐ A BPEL engine has several activities designed to perform the process logic, including WaitRT, AssignRT, Flow RT and SequenceRT.
- ☐ Among those activities, FlowRT and SequenceRT can contain multiple activities.



Requirements Statements₁

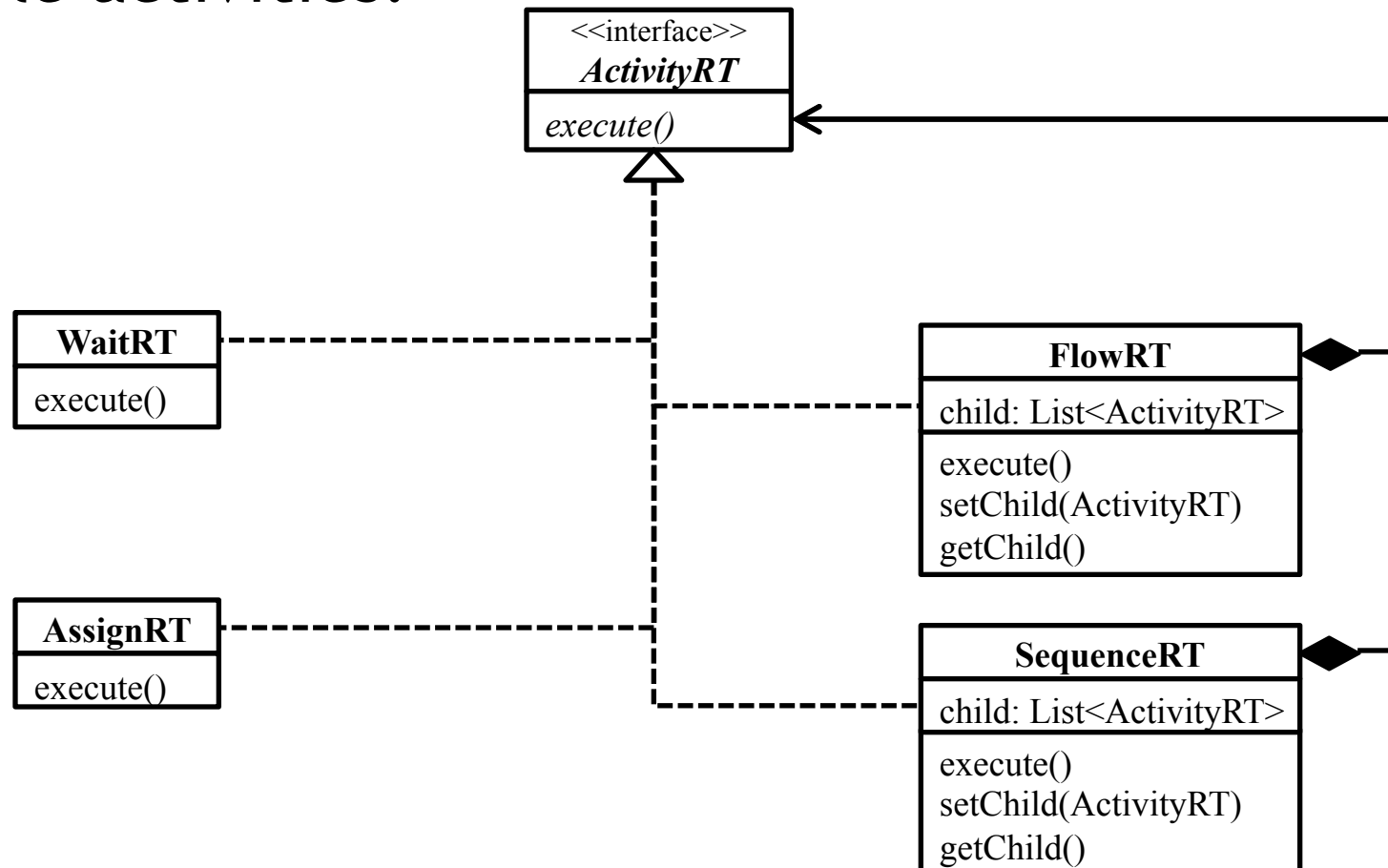
- A BPEL engine has several activities designed to perform the process logic, including WaitRT, AssignRT, Flow RT and SequenceRT.





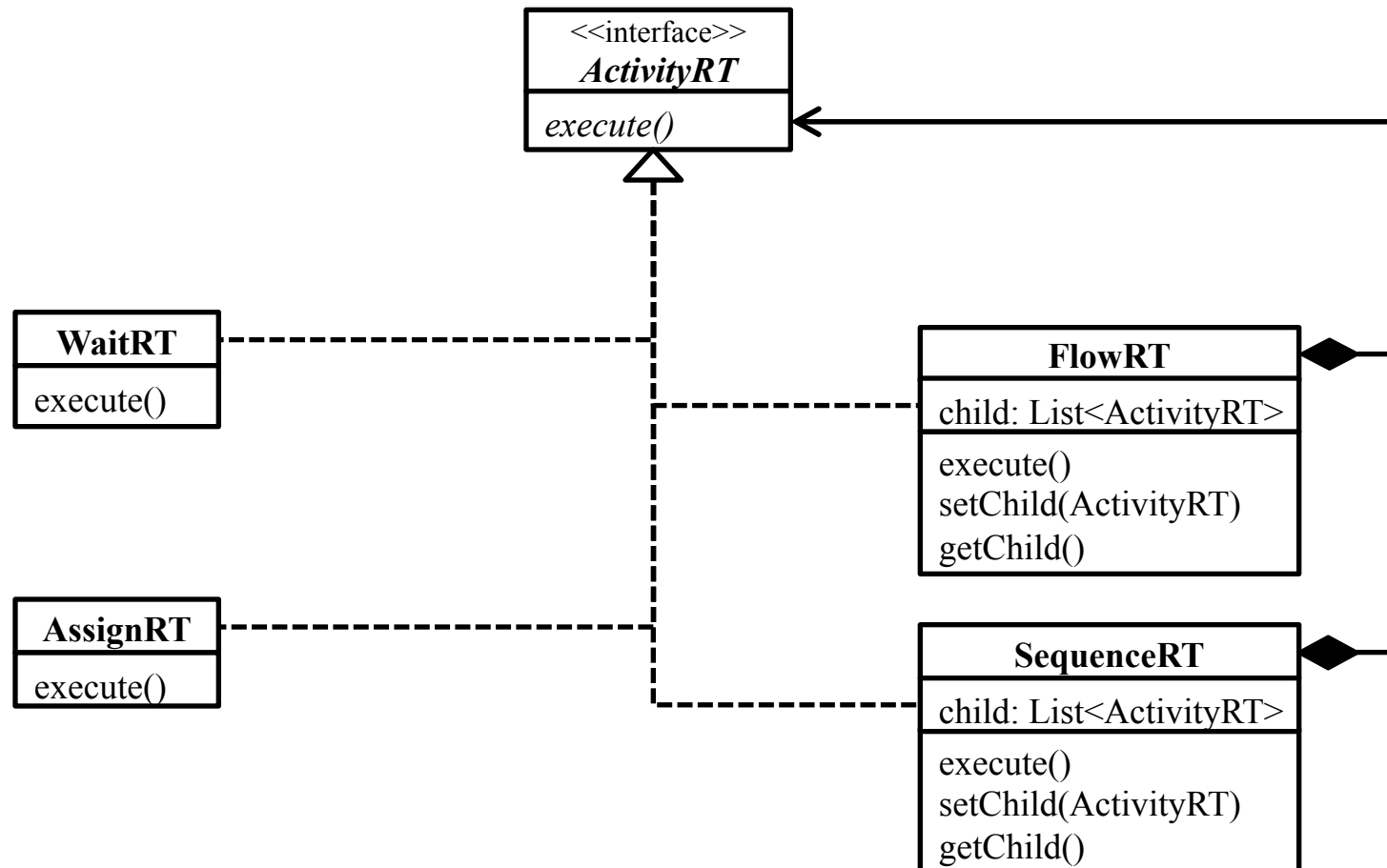
Requirements Statements₂

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

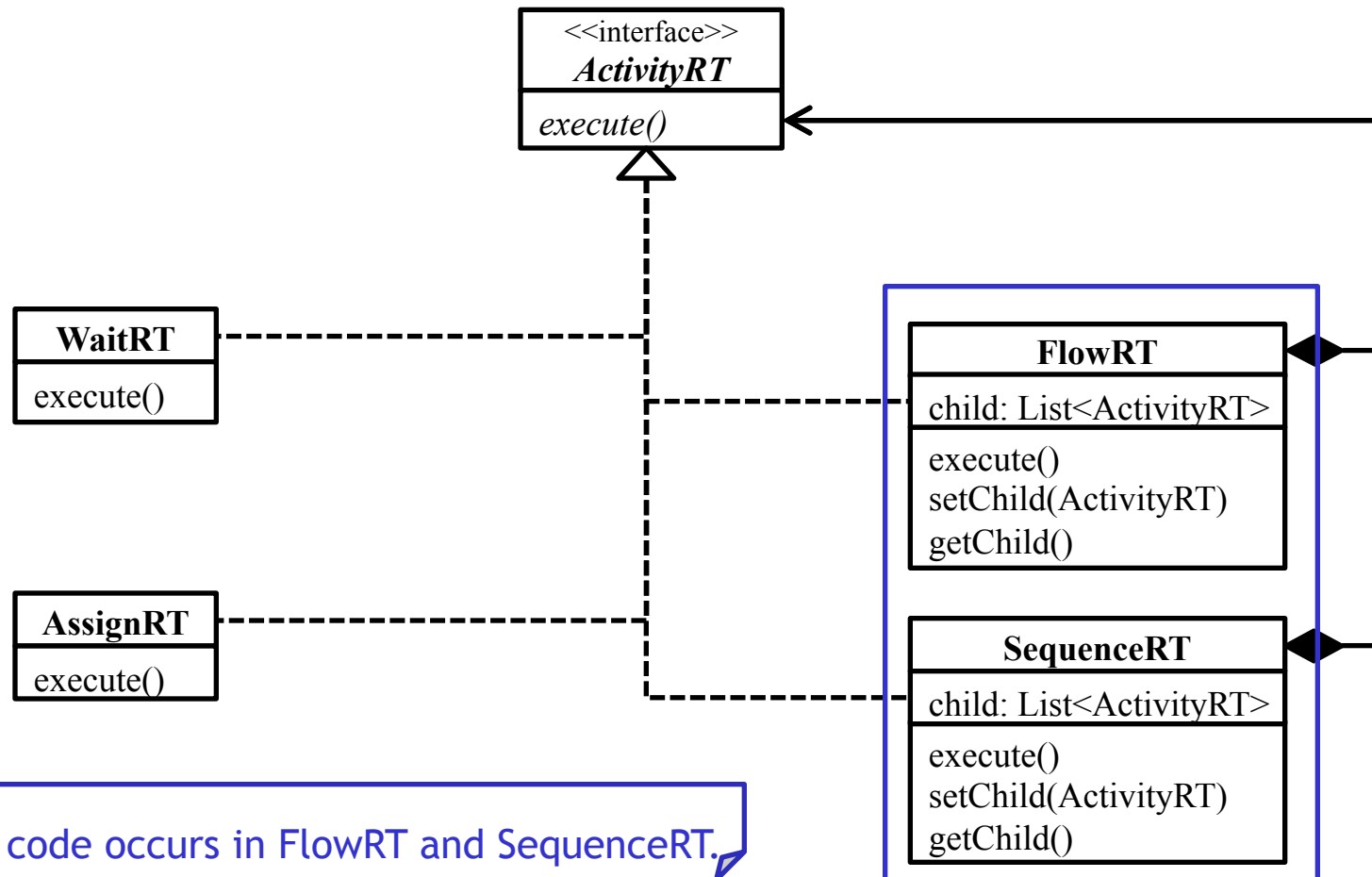




Initial Design

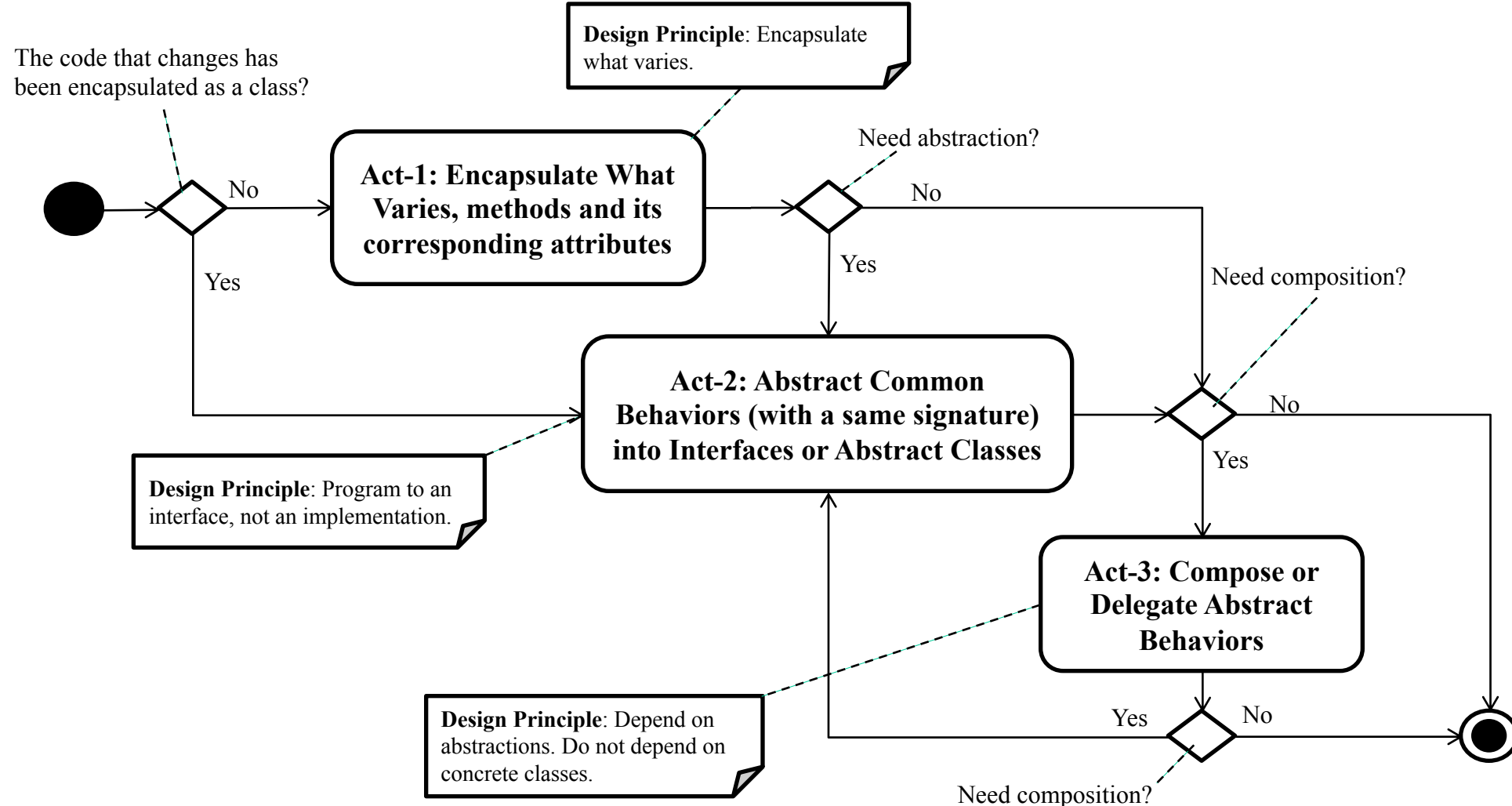


The Problem with the Initial Design

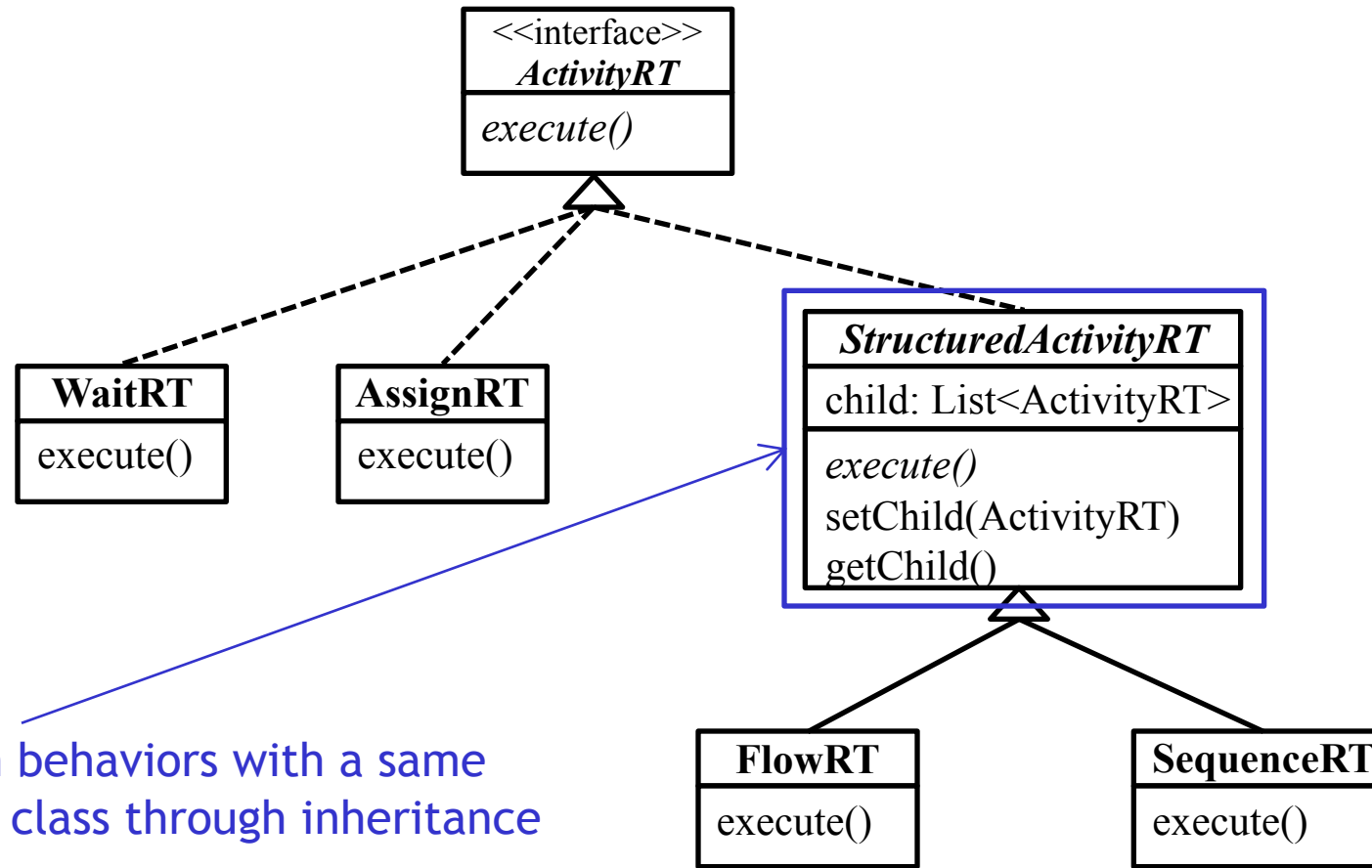


Problem: Duplicate code occurs in FlowRT and SequenceRT.

Design Process for Change



Act-2: Abstract Common Behaviors

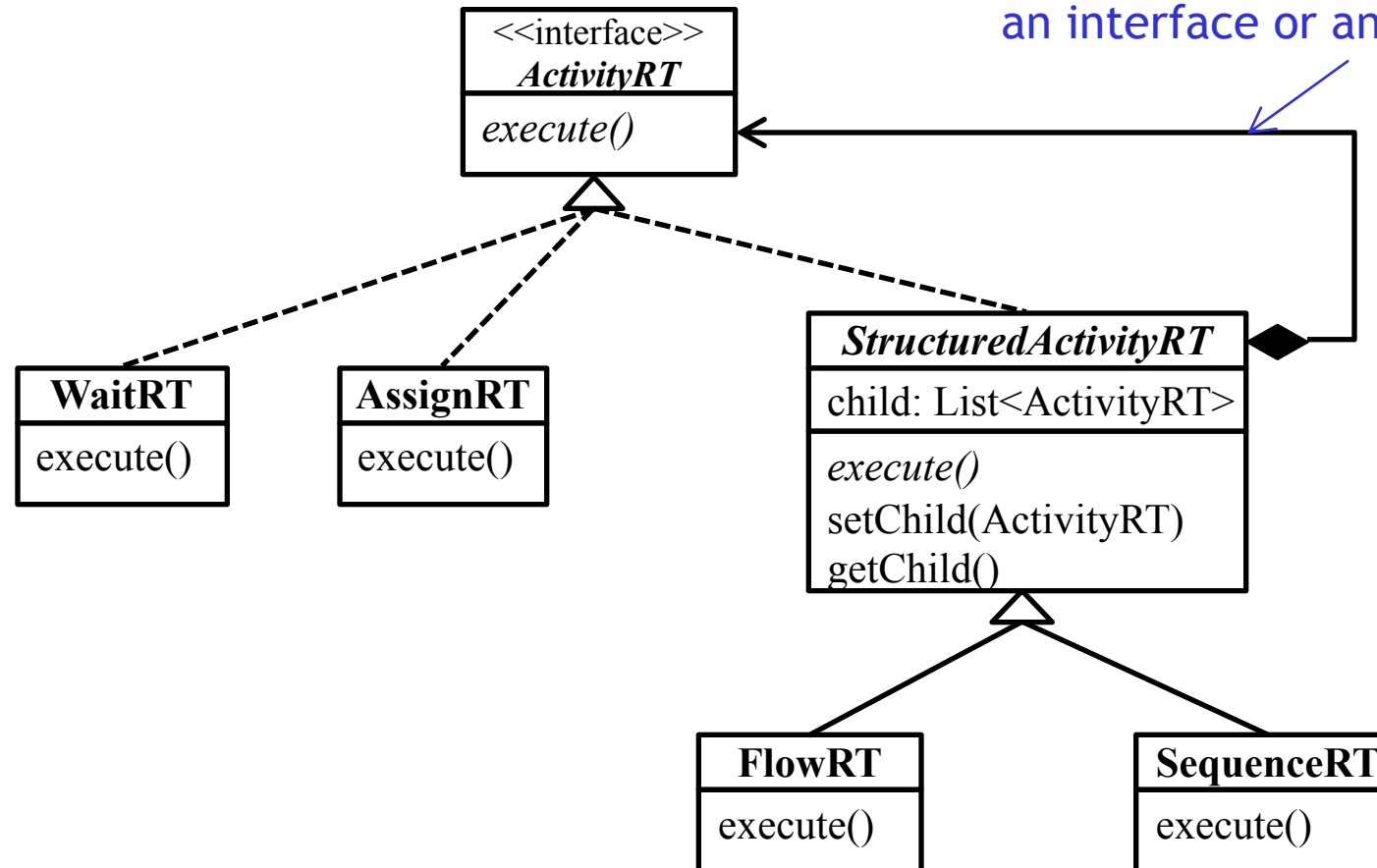


Act-2.1: Abstract common behaviors with a same signature into an abstract class through inheritance



Act-3: Compose Abstract Behaviors

Act-3.1: Compose behaviors of an interface or an abstract class





Refactored Design after Design Process

