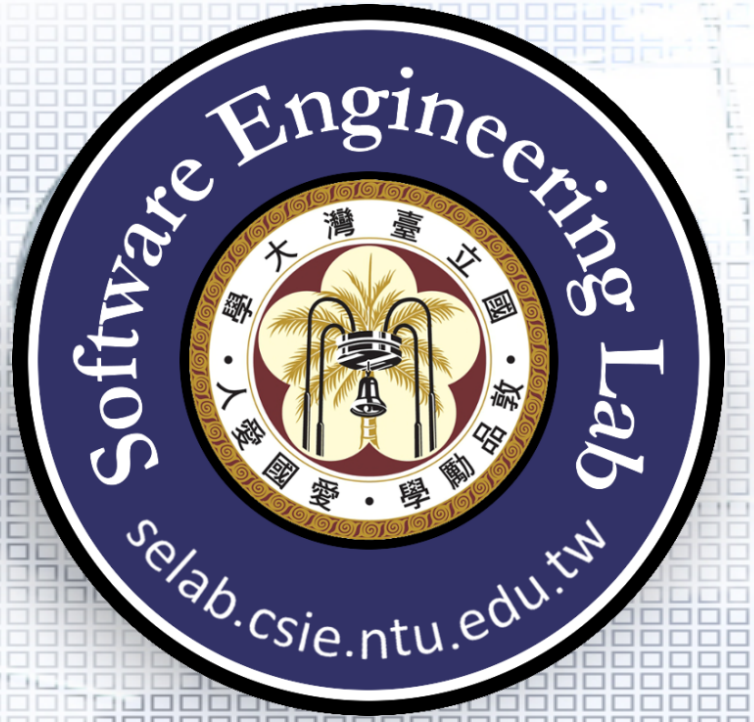
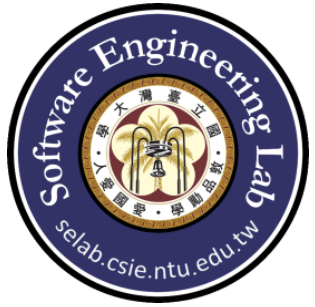


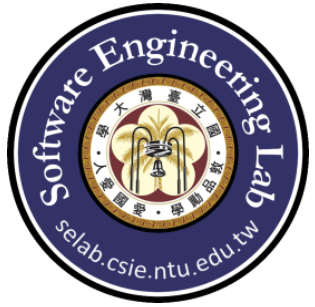
Abstract Factory Pattern

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



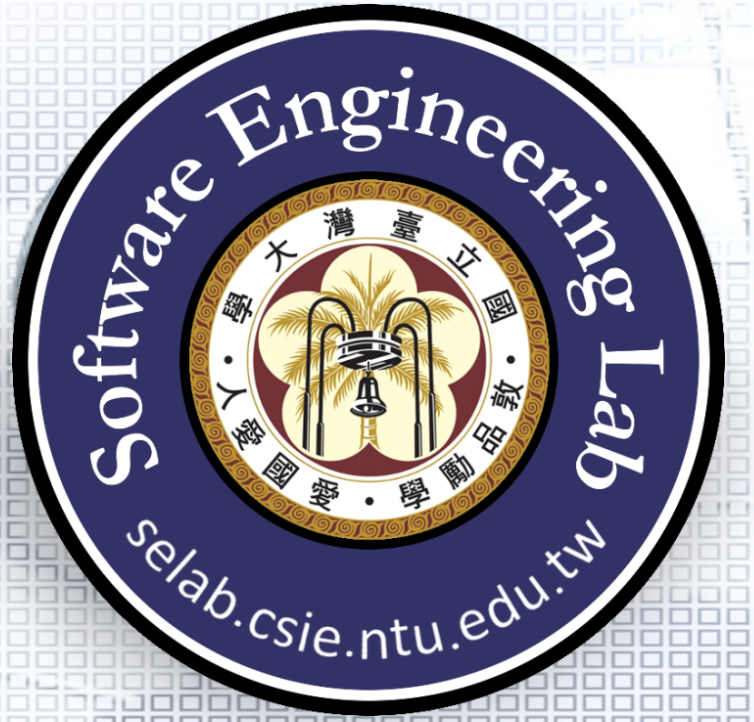
Design Aspect of Abstract Factory

Families of product objects



Outline

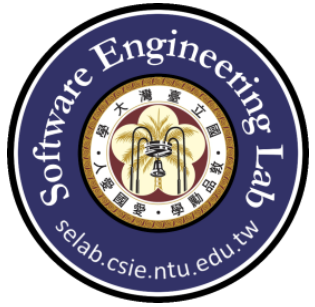
- A GUI Application with Multiple Styles Requirements Statements
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- Common Problems
- Intent
- Abstract Factory Pattern Structure
- Abstract Factory vs. Factory Method
- Pizza Store (Extended): Another Example
- Homework



A GUI Application with Multiple Styles (Abstract Factory)

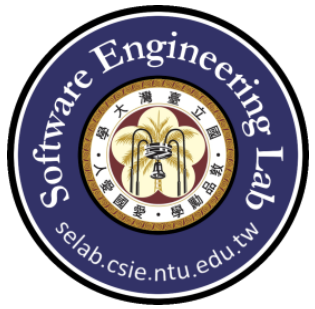
Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University



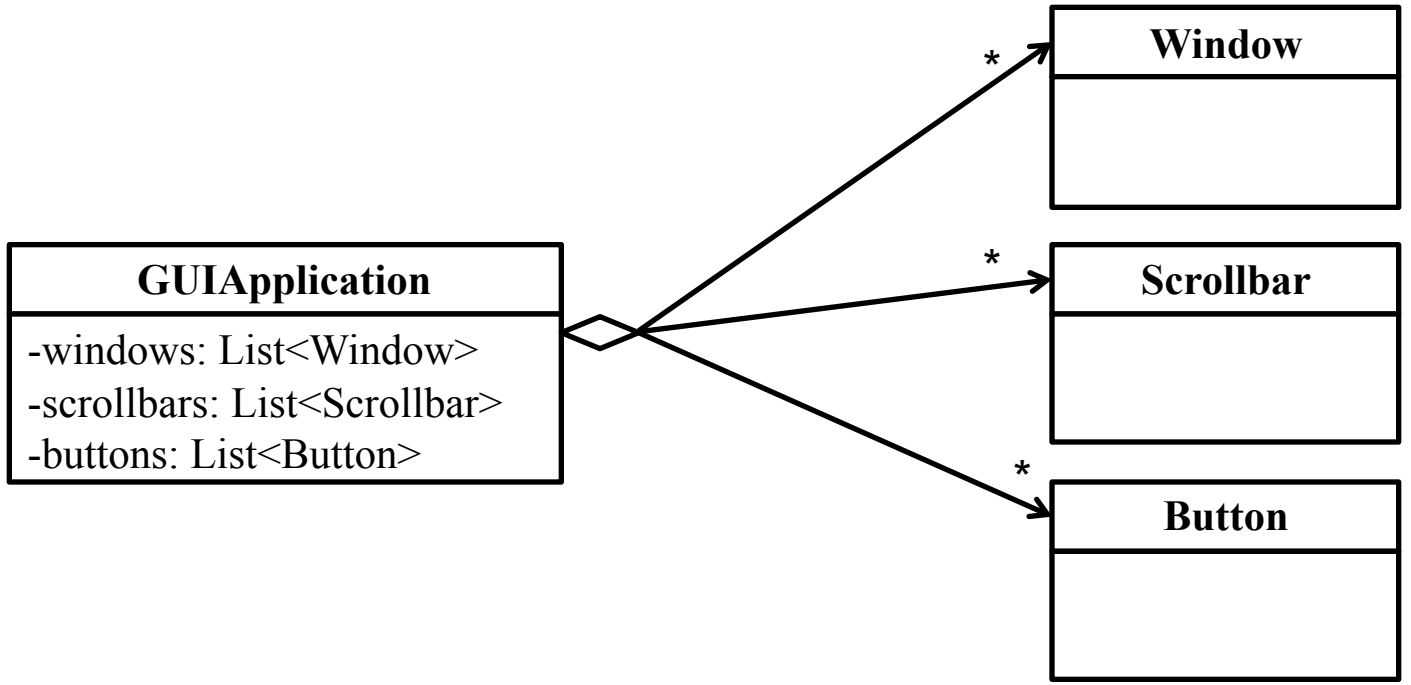
Homework: Requirements Statements

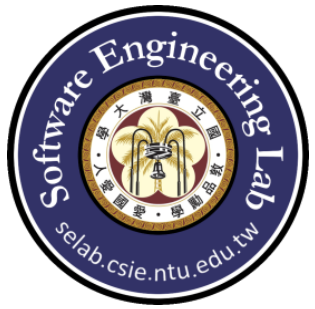
- A GUI Application consists of various types of widgets such as window, scroll bar, and button.
- Each widget in the GUI application has two or more implementations according to different look-and-feel standards, such as Motif and Presentation Manager.
- The GUI application can switch its look-and-feel style from one to another while the widgets are being created.



Requirements Statements$_1$

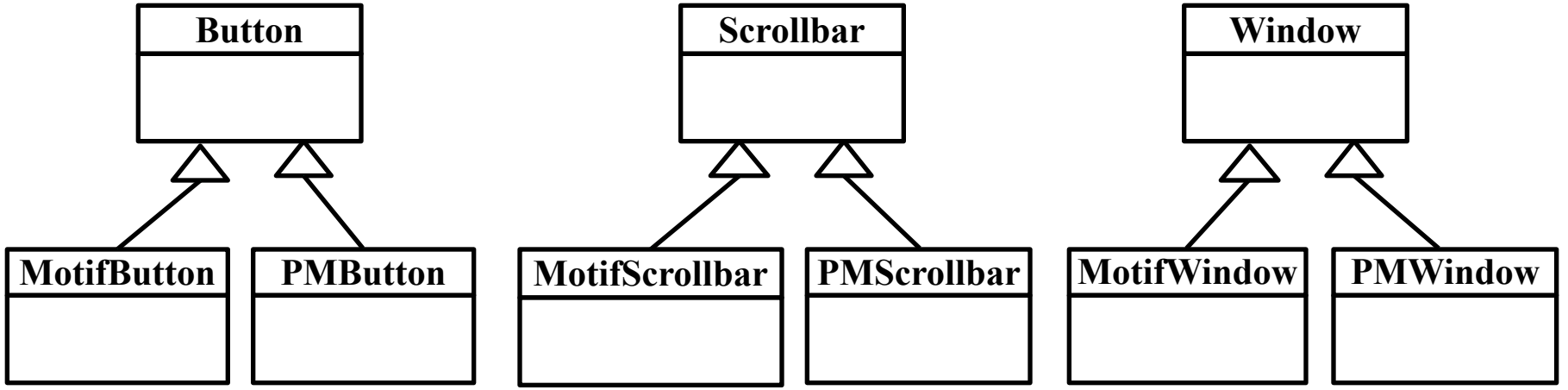
- A GUI Application consists of various types of widgets such as window, scroll bar, and button.





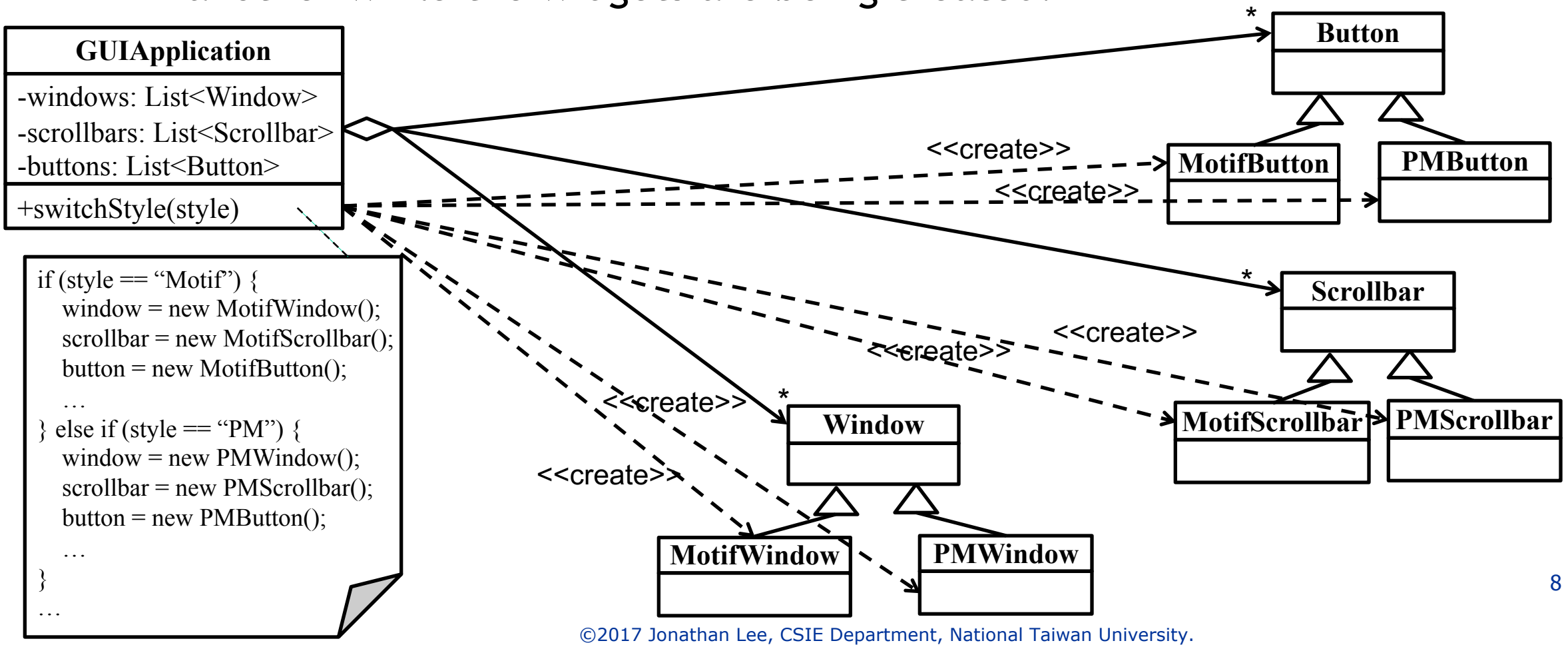
Requirements Statements$_2$

- Each widget in the GUI application has two or more implementations according to different look-and-feel standards, such as Motif and Presentation Manager.

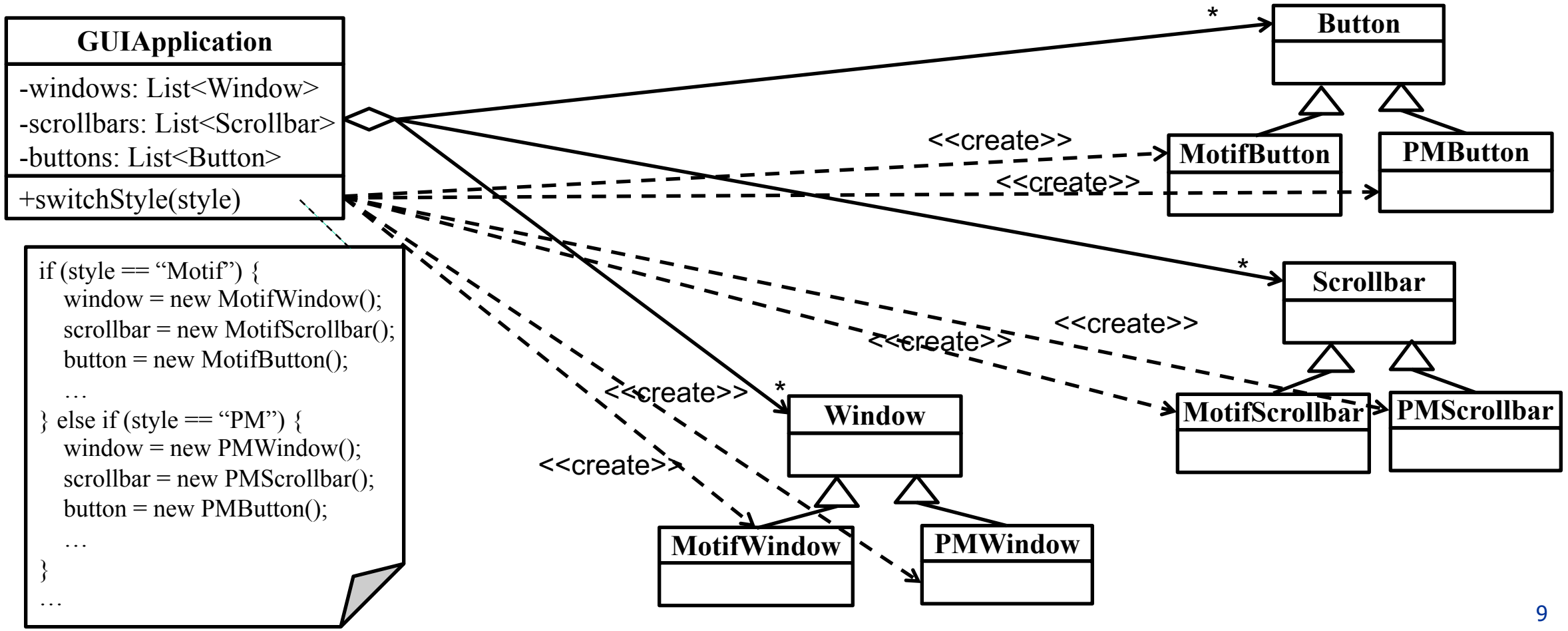


Requirements Statements$_3$

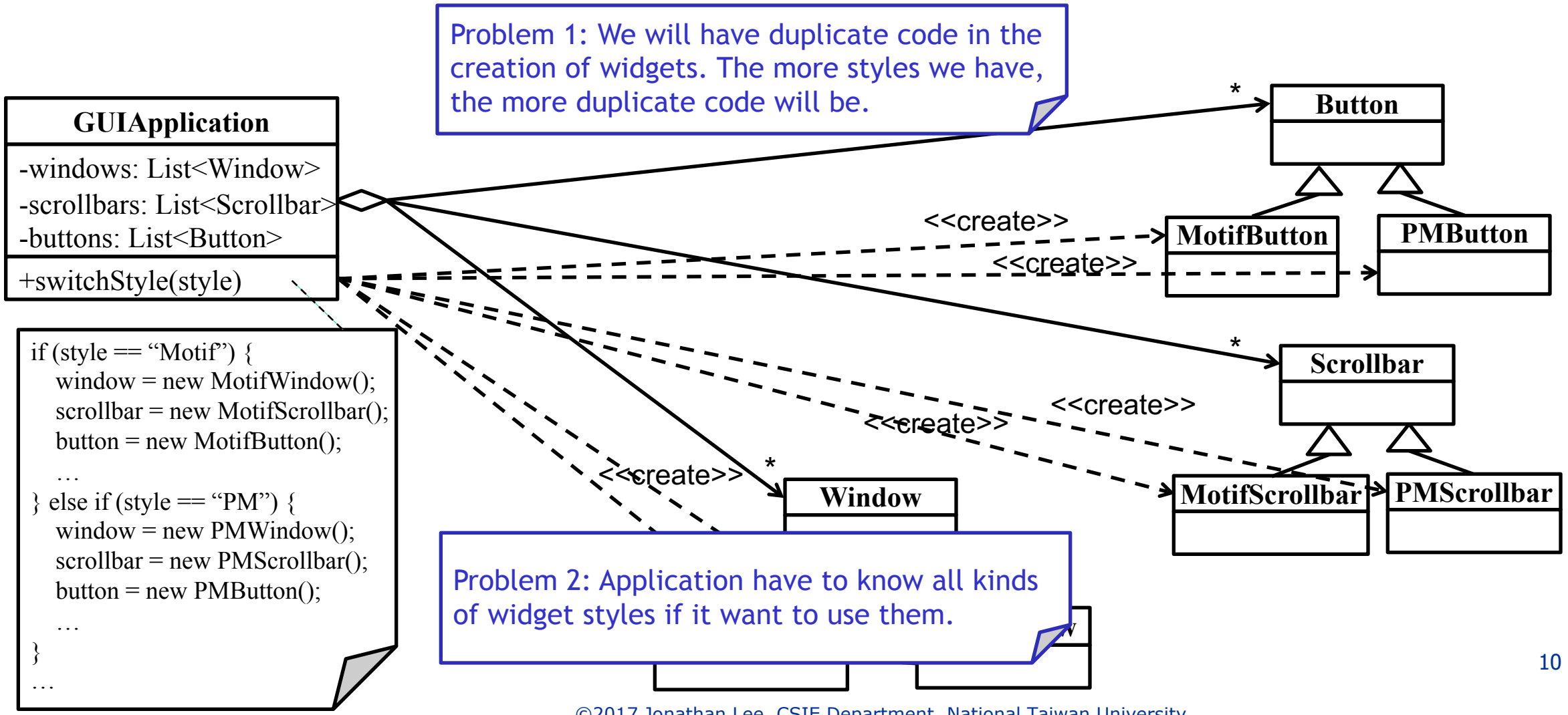
- The GUI application can switch its look-and-feel style from one to another while the widgets are being created.



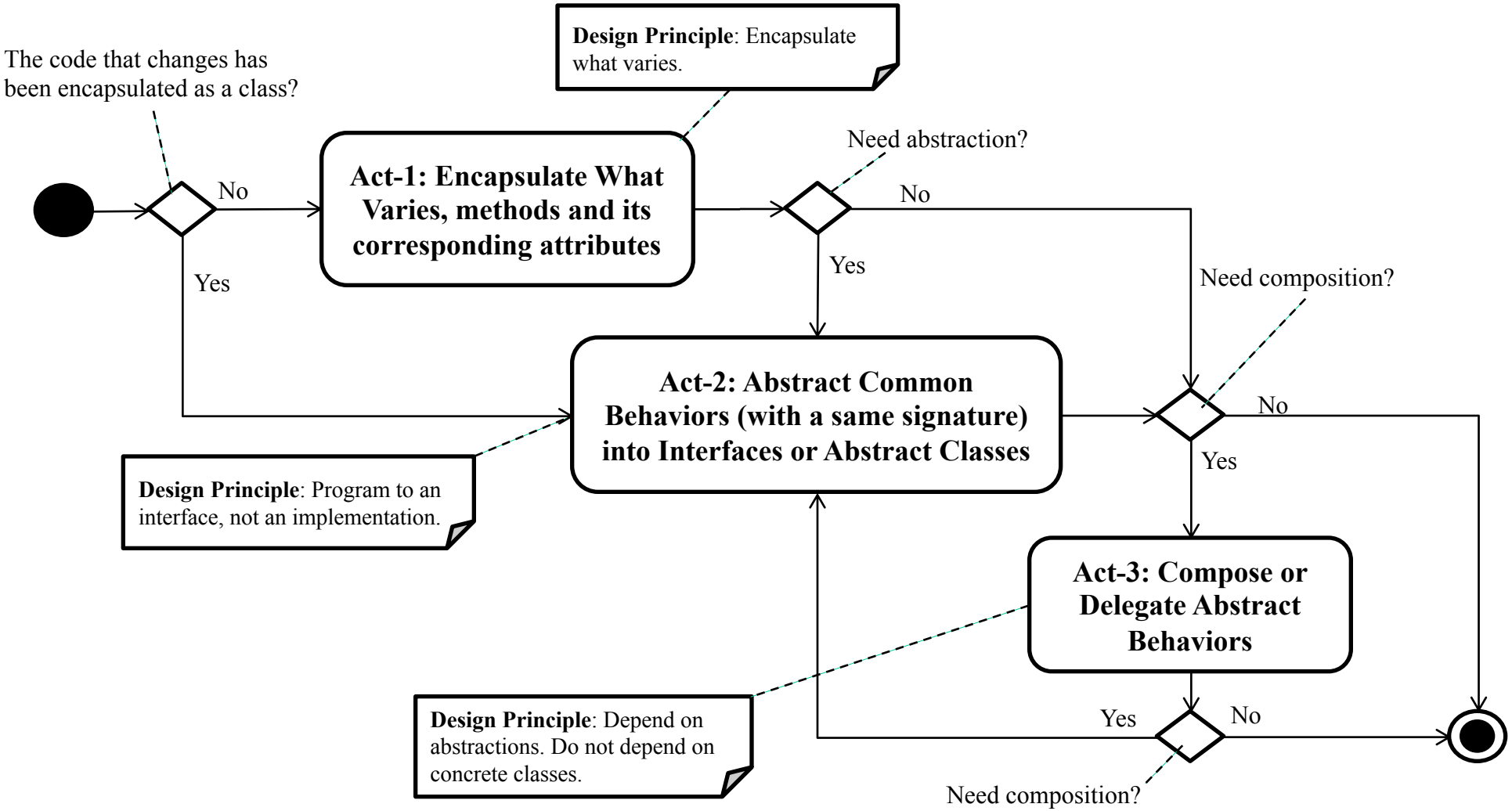
Initial Design



Problems with Initial Design

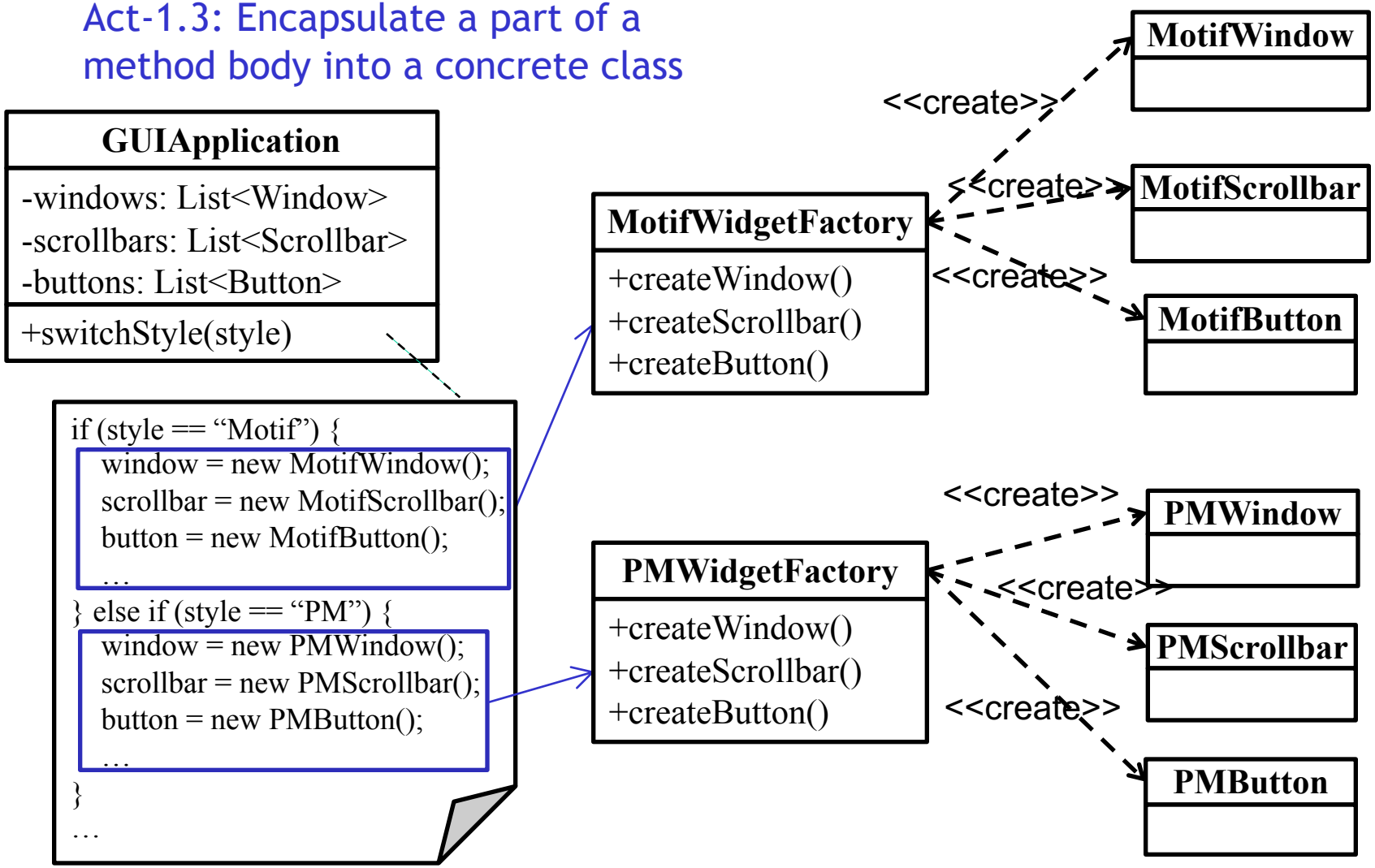


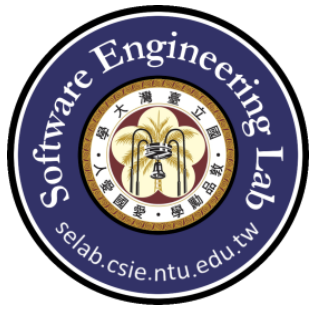
Design Process for Change



Act-1: Encapsulate What Varies

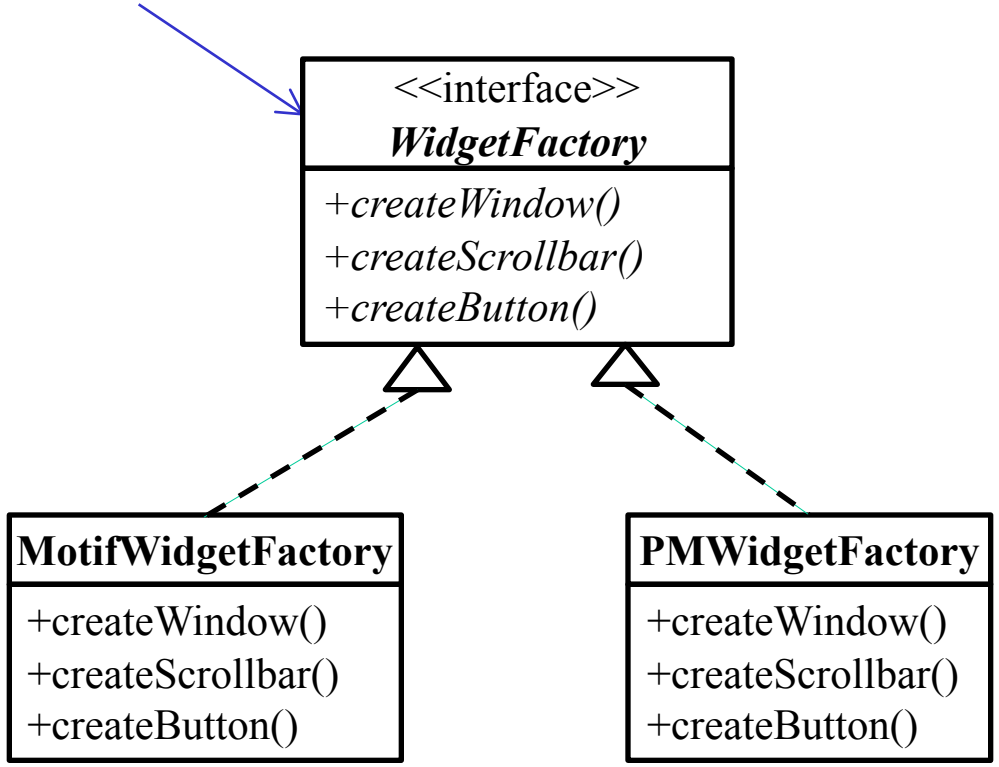
Act-1.3: Encapsulate a part of a method body into a concrete class



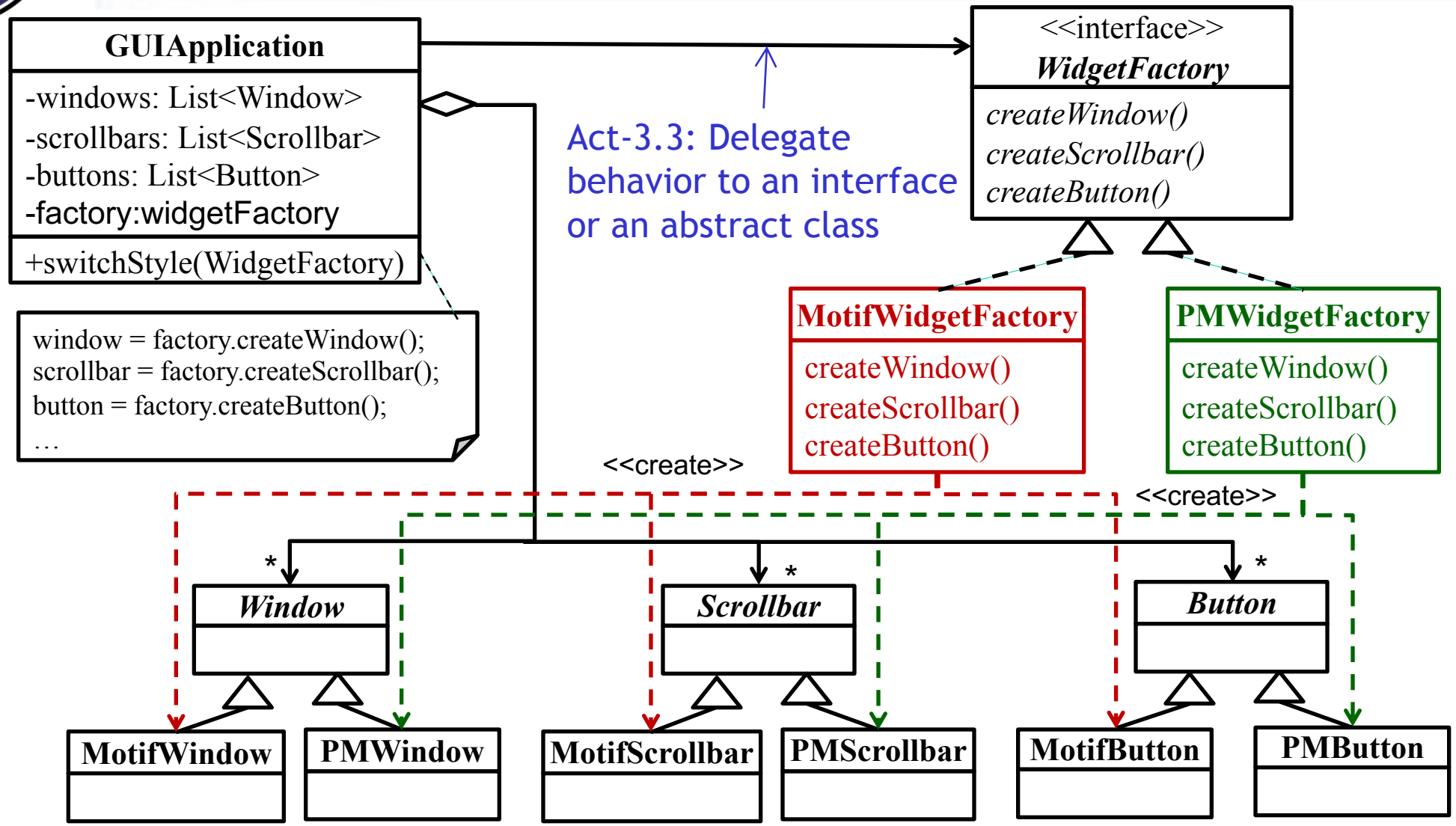


Act-2: Abstract Common Behaviors

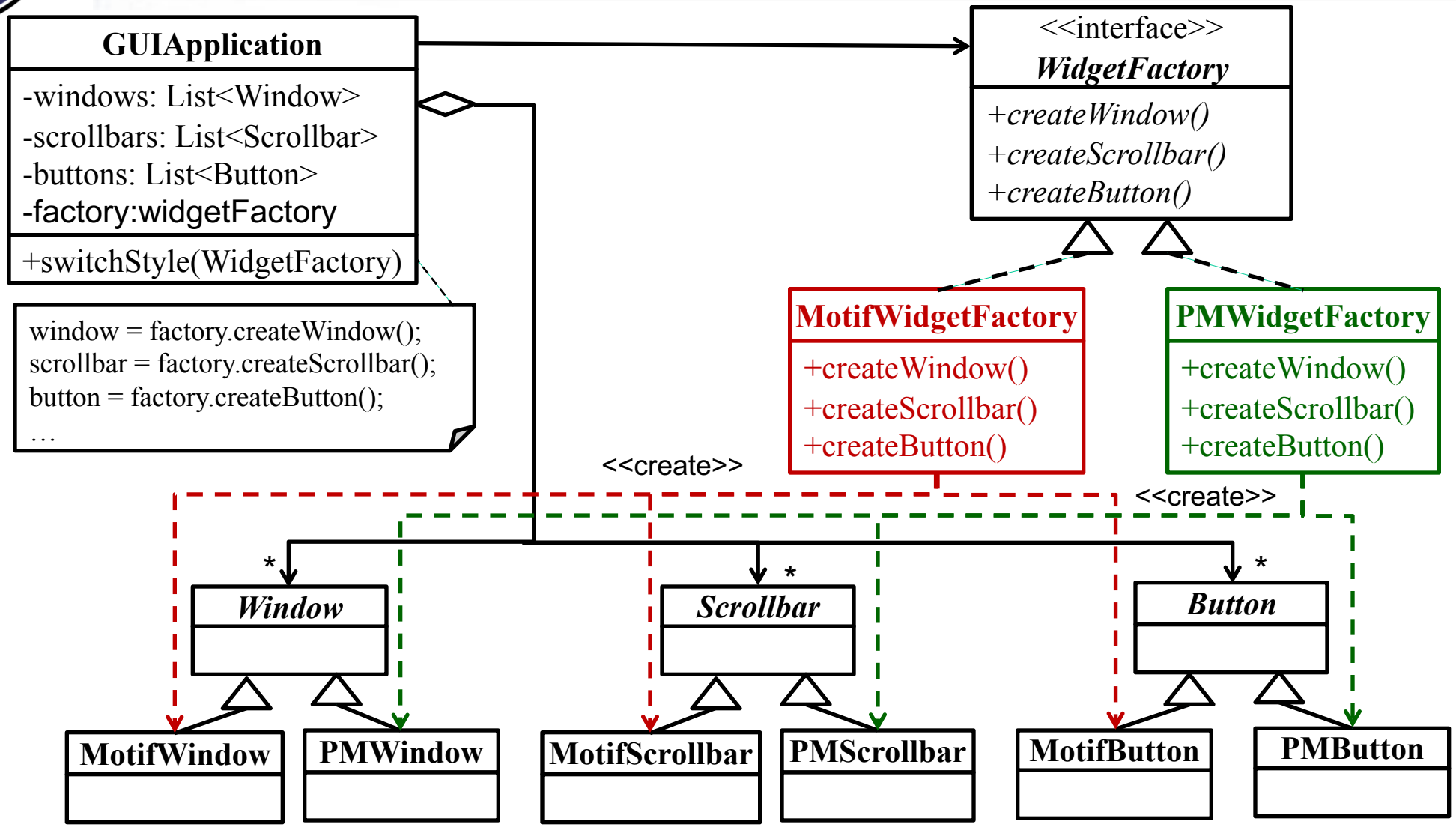
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

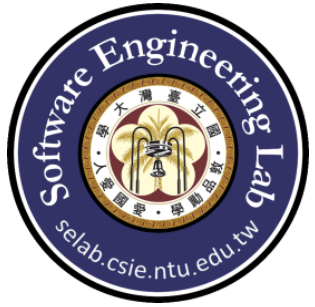


Act-3: Compose Abstract Behaviors



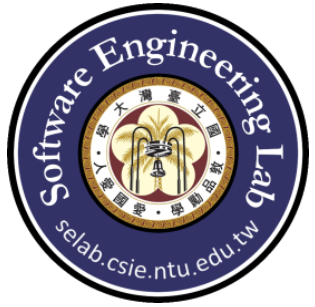
Refactored Design after Design Process





GUIApplication Class (1)

```
public class GUIApplication {
    private List<Button> buttons = new ArrayList<>();
    private List<Window> windows = new ArrayList<>();
    private List<ScrollBar> scrollBars = new ArrayList<>();
    private WidgetFactory widgetFactory = new MotifWidgetFactory();

    public void swtichStyle(WidgetFactory factory){
        this.widgetFactory = factory;

        List<Button> tempButtonList = new ArrayList<>();
        for(Button button: buttons){
            tempButtonList.add(factory.createButton(button.getName()));
        }
        buttons.clear();
        buttons.addAll(tempButtonList);
        tempButtonList.clear();


        List<Window> tempWindowList = new ArrayList<>();
        for(Window window: windows){
            tempWindowList.add(factory.createWindow(window.getName()));
        }
        windows.clear();
        windows.addAll(tempWindowList);
        tempWindowList.clear();


        List<ScrollBar> tempScrollBarList = new ArrayList<>();
        for(ScrollBar scrollBar: scrollBars){
            tempScrollBarList.add(factory.createScrollBar(scrollBar.getName()));
        }
        scrollBars.clear();
        scrollBars.addAll(tempScrollBarList);
        tempScrollBarList.clear();
    }
```



Source code

Window

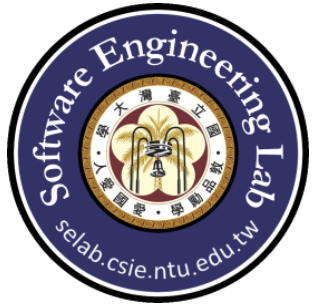
```
public class Window {
    private String name;

    public Window(String name) { this.name = name; }

    public String getName() { return name; }
}
```

PMWindow

```
public class PMWindow extends Window{

    public PMWindow(String name) { super(name); }
}
```

MotifWindow

```
public class MotifWindow extends Window{
    public MotifWindow(String name) {
        super(name);
    }
}
```



Source code

ScrollBar

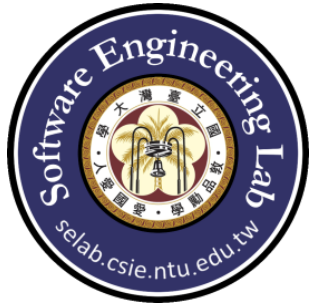
```
public class ScrollBar {
    private String name;

    public ScrollBar(String name) { this.name = name; }

    public String getName() { return name; }
}
```

PMScrollBar

```
public class PMScrollBar extends ScrollBar{

    public PMScrollBar(String name) { super(name); }

}
```

MotifScrollBar

```
public class MotifScrollBar extends ScrollBar{

e package-private more... (⌘F1)r(String name) { super(name); }

}
```



Source code

Button

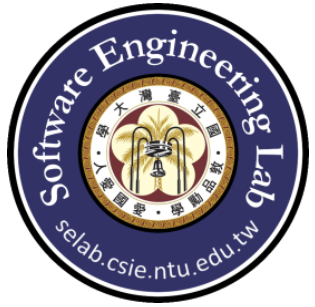
```
public class Button {
    private String name;

    public Button(String name) { this.name = name; }

    public String getName() { return name; }
}
```

PMButton

```
public class PMButton extends Button{

    public PMButton(String name) { super(name); }

}
```

MotifButton

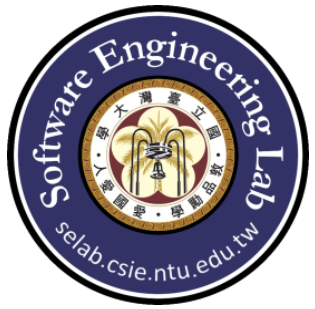
```
public class MotifButton extends Button{

    public MotifButton(String name) { super(name); }

}
```



Source code

WidgetFactory

```
public interface WidgetFactory {

    public Window createWindow(String name);

    public ScrollBar createScrollBar(String name);

    public Button createButton(String name);
}
```



Source code

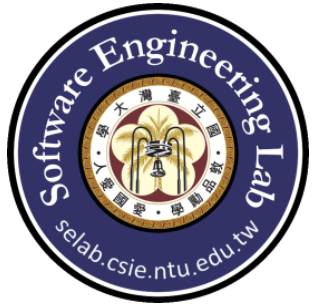
MotifWidgetFactory

```
public class MotifWidgetFactory implements WidgetFactory{

    @Override
    public Window createWindow(String name) {
        // TODO Auto-generated method stub
        return new MotifWindow(name);
    }

    @Override
    public ScrollBar createScrollBar(String name) {
        // TODO Auto-generated method stub
        return new MotifScrollBar(name);
    }

    @Override
    public Button createButton(String name) {
        // TODO Auto-generated method stub
        return new MotifButton(name);
    }

}
```

PMWidgetFactory

```
public class PMWidgetFactory implements WidgetFactory{

    @Override
    public Window createWindow(String name) {
        // TODO Auto-generated method stub
        return new PMWindow(name);
    }

    @Override
    public ScrollBar createScrollBar(String name) {
        // TODO Auto-generated method stub
        return new PMScrollBar(name);
    }

    @Override
    public Button createButton(String name) {
        // TODO Auto-generated method stub
        return new PMButton(name);
    }

}
```

TODO Auto-generated method stub



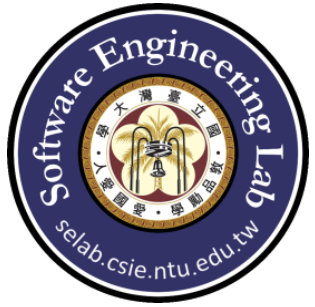
Input / Output

Input:

```
[Widget_type] [Widget_name]//add widget

[look-and-feel_style]/*set [look-and-feel_style] as current
style*/

Present /*extra command, show all widgets to standard output*/

...
```

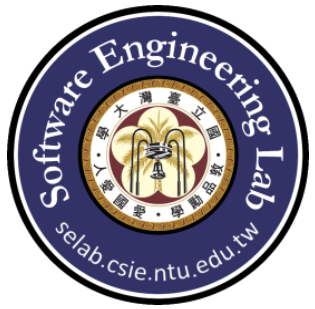
Output:

```
/*

Widgets must be shown with following rules:

     Window should be shown before ScrollBar.

     ScrollBar should be shown before Button.

If there are the same type widgets, show with the sequential
order from input.
*/

[Style_widget_type] [Widget_name]

...
```



Test cases

- TestCase 1: Three kinds of widget
- TestCase 2: Same kind of widget has more than one.
- TestCase 3: SwtichStyle, PM and Motif
- TestCase 4: Only Button
- TestCase 5: Only Window
- TestCase 6: Only Scroll Bar
- TestCase 7: Complex



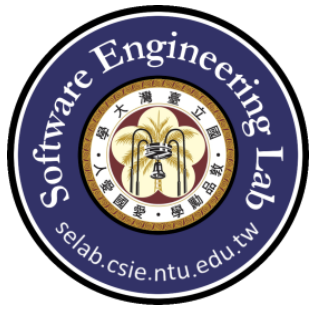
Test case1

Sample1.in

```
1  Button button1
2  Window window1
3  ScrollBar scrollBar1
4  Present
5
```

Sample1.out

```
1  MotifWindow window1
2  MotifScrollBar scrollBar1
3  MotifButton button1
```



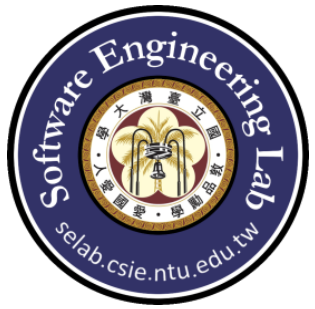
Test case2

Sample2.in

```
1  Button button1
2  Button button2
3  Window window1
4  Window window2
5  ScrollBar scrollBar1
6  ScrollBar scrollBar2
7  Present
```

Sample2.out

```
1  MotifWindow window1
2  MotifWindow window2
3  MotifScrollBar scrollBar1
4  MotifScrollBar scrollBar2
5  MotifButton button1
6  MotifButton button2
```



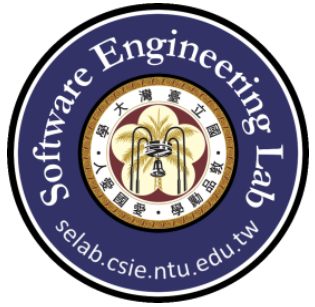
Test case3

Sample3.in

```
1  Button button1
2  Window window1
3  ScrollBar scrollBar1
4  Present
5  PM
6  Present
7  Motif
8  Present
```

Sample3.out

```
1  MotifWindow window1
2  MotifScrollBar scrollBar1
3  MotifButton button1
4  PMWindow window1
5  PMScrollBar scrollBar1
6  PMButton button1
7  MotifWindow window1
8  MotifScrollBar scrollBar1
9  MotifButton button1
```



Test cases

Test case4

Sample4.in

```
1 Button button1
2 Present
3
```

Sample4.out

```
1 MotifButton button1
2
```

Test case5

Sample5.in

```
1 Window window1
2 Present
```

Sample5.out

```
1 MotifWindow window1
```

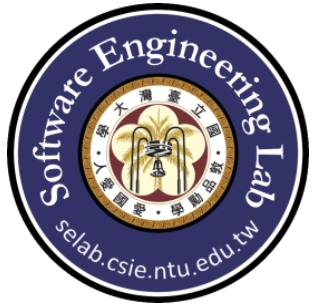
Test case6

Sample6.in

```
1 ScrollBar scrollBar1
2 Present
```

Sample6.out

```
1 MotifScrollBar scrollBar1
```



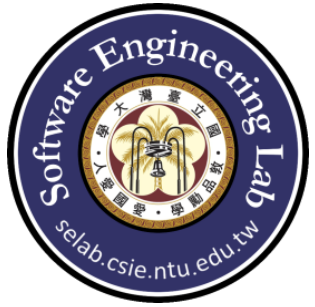
Test case7

Sample7.in

```
1  PM
2  Present
3  Button button1
4  Button button2
5  Window window1
6  Window window2
7  Motif
8  ScrollBar scrollBar1
9  ScrollBar scrollBar2
10 Present
11 PM
12 ScrollBar scrollBar3
13 Present
14 Window window3
15 Motif
16 PM
17 Present
18 Window window4
19 Present
```

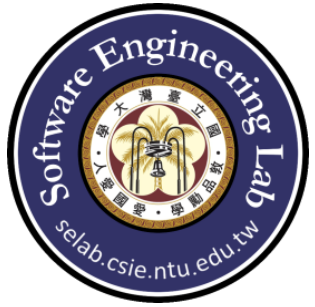
Sample7.out

```
1  MotifWindow window1
2  MotifWindow window2
3  MotifScrollBar scrollBar1
4  MotifScrollBar scrollBar2
5  MotifButton button1
6  MotifButton button2
7  PMWindow window1
8  PMWindow window2
9  PMScrollBar scrollBar1
10 PMScrollBar scrollBar2
11 PMScrollBar scrollBar3
12 PMButton button1
13 PMButton button2
14 PMWindow window1
15 PMWindow window2
16 PMWindow window3
17 PMScrollBar scrollBar1
18 PMScrollBar scrollBar2
19 PMScrollBar scrollBar3
20 PMButton button1
21 PMButton button2
22 PMWindow window1
23 PMWindow window2
24 PMWindow window3
25 PMWindow window4
26 PMScrollBar scrollBar1
27 PMScrollBar scrollBar2
28 PMScrollBar scrollBar3
29 PMButton button1
30 PMButton button2
```



Recurrent Problem

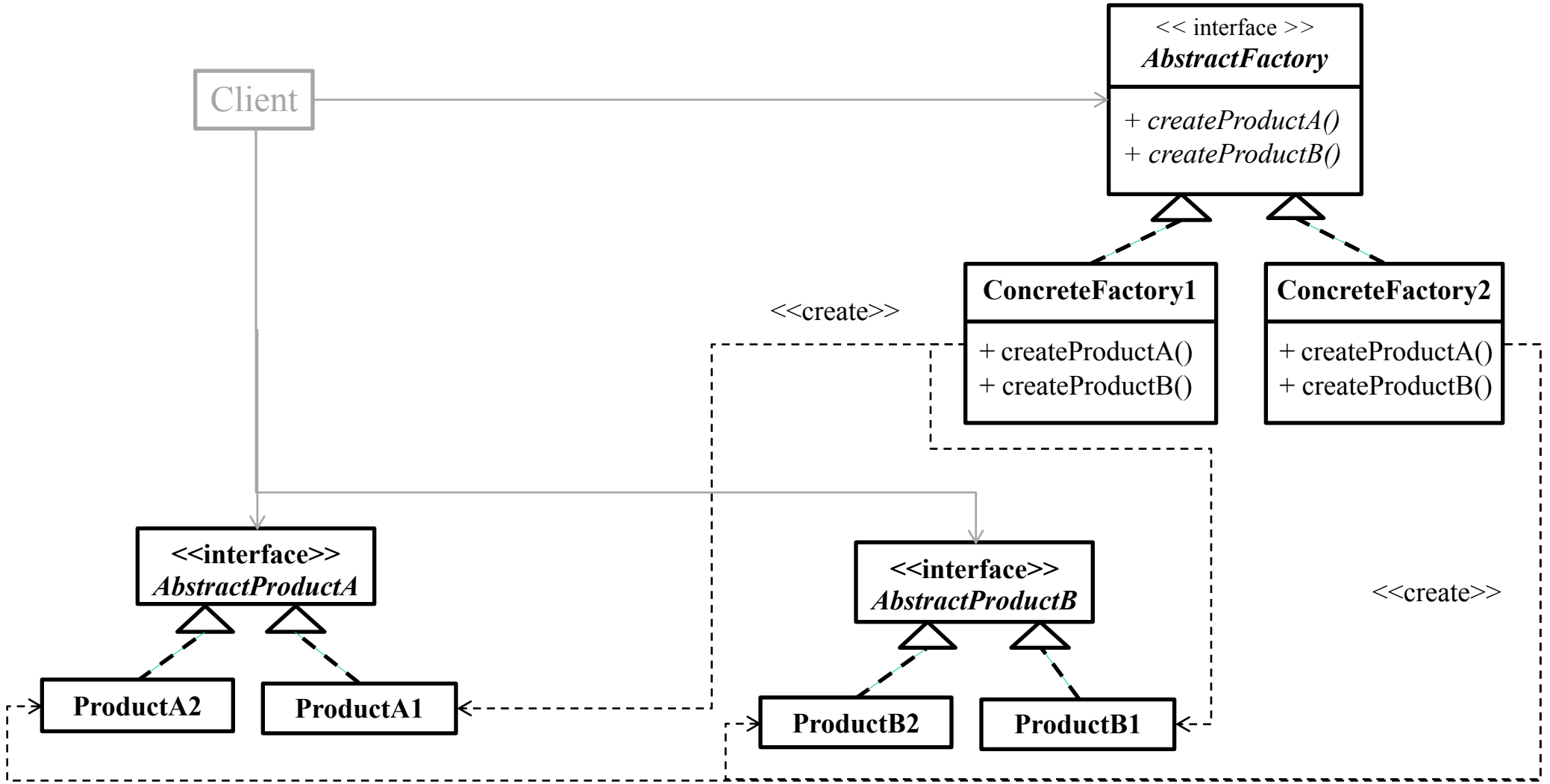
- As the families of related or dependent objects are added, we need to write new object classes for the new families and return the product immediately.
  - For example, different look-and-feels define different appearances and behaviors for user interface "widgets" like scroll bars, windows, and buttons.

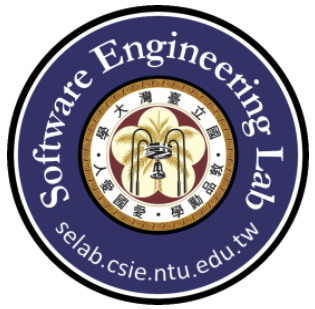


Intent

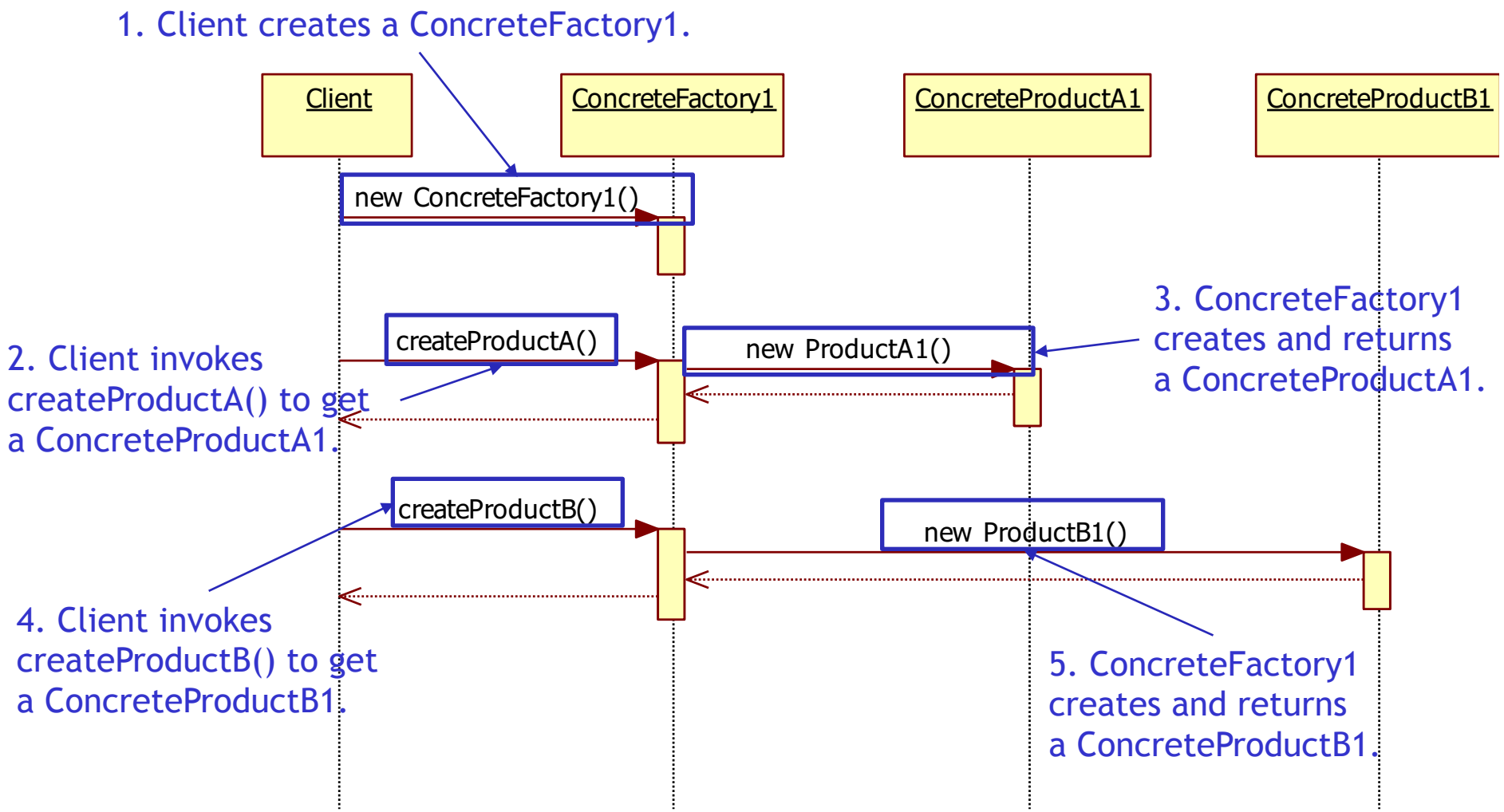
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

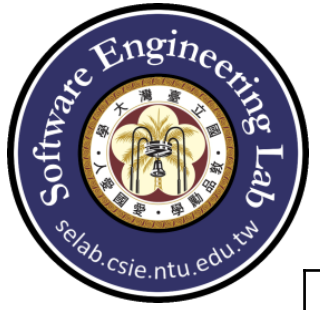
Abstract Factory Structure$_1$





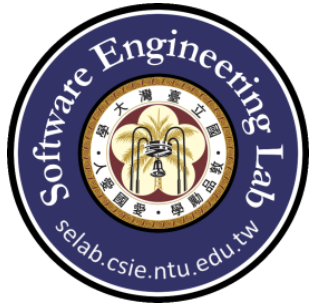
Abstract Factory Structure$_2$





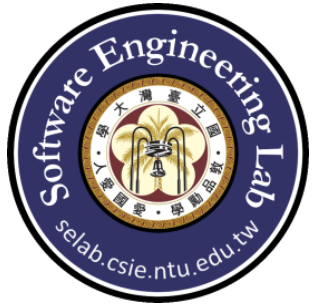
Abstract Factory Structure$_3$

	Instantiation	Use	Termination
Client	Other class except classes in the AbstractFactory	Other class except classes in the Abstract Factory	Other class except classes in the AbstractFactory
Abstract Factory	X	Client class uses this interface to get a product which is produced by ConcreteFactory through polymorphism	X
Concrete Factory	Other class or the client class	Client class uses this class to get a product through AbstractFactory	Other class or the client class
Abstract ProductA	X	Client class uses ConcreteProductA through this interface	X
Concrete ProductA	ConcreteFactory	Client class	Other class or the client class
Abstract ProductB	X	Client class uses ConcreteProductB through this interface	X
Concrete ProductB	ConcreteFactory	Client class	Other class or the client class



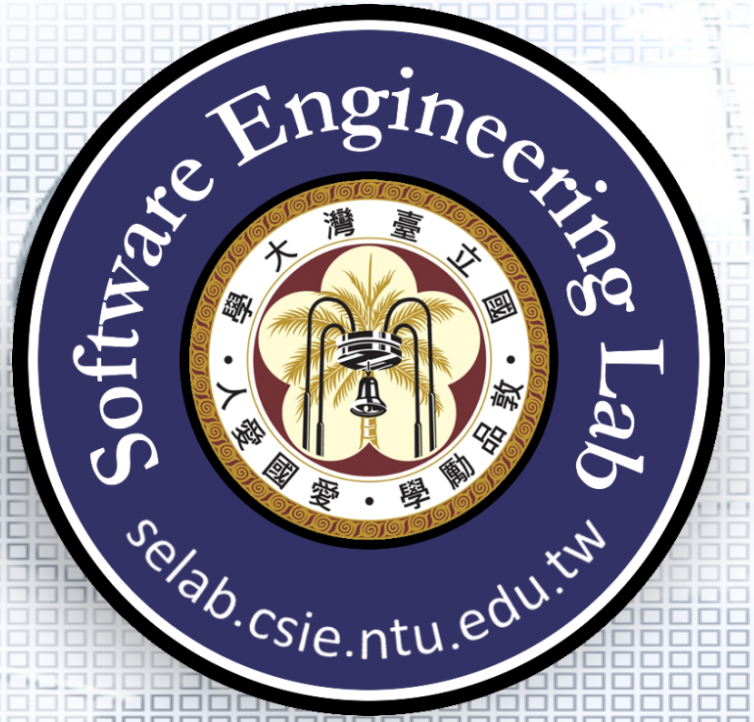
Direct Creation vs. Indirect Creation

- Direct Creation
  - Create an object with its constructor
  - Once the creation is changed, the class should be modified
  - E.g. Sauce sauce = new MarinaraSauce();
- Indirection
  - Delegate to other class for creation
  - If the creation is changed or extended, just add another creator class instead of modifying the class.
  - E.g. Factory Method, Abstract Factory, Builder and etc.



Abstract Factory vs. Factory Method

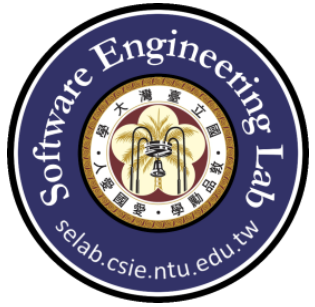
- Factory Method
  - creates a single product
- Abstract Factory
  - consists of multiple factory methods
  - each factory method creates a related or dependent product



Extended Pizza Store (Abstract Factory)

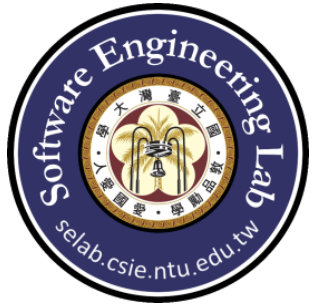
Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University



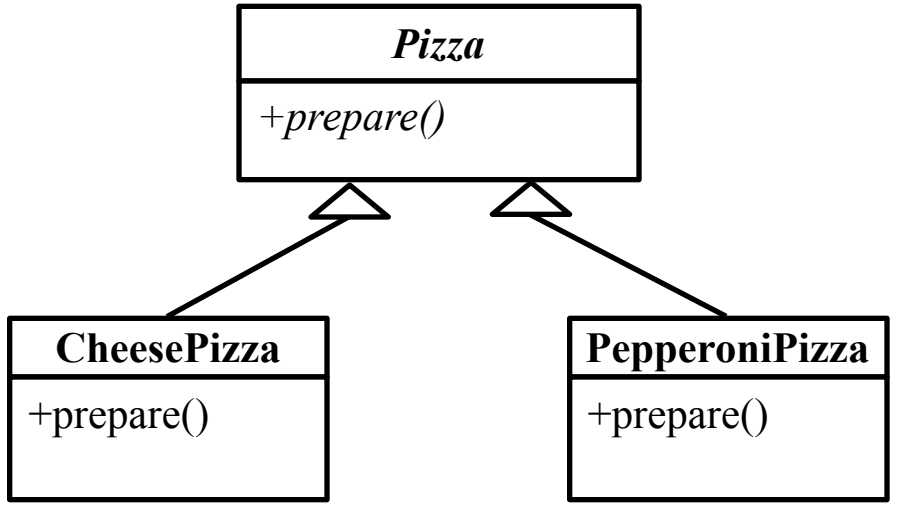
Requirements Statements

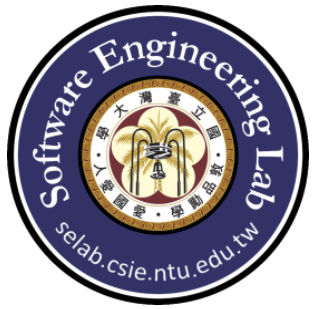
- In a pizza store system, two flavors of pizza are offered: Cheese Pizza and Pepperoni Pizza.
- Each flavor of pizza can be categorized into two styles: New York Style and Chicago Style.
- Each pizza style requires its own type of dough and sauce, for example,
  - NY Style: Thin Crust Dough, Marinara Sauce
  - Chicago Style: Thick Crust Dough, Plum Tomato Sauce



Requirements Statements$_1$

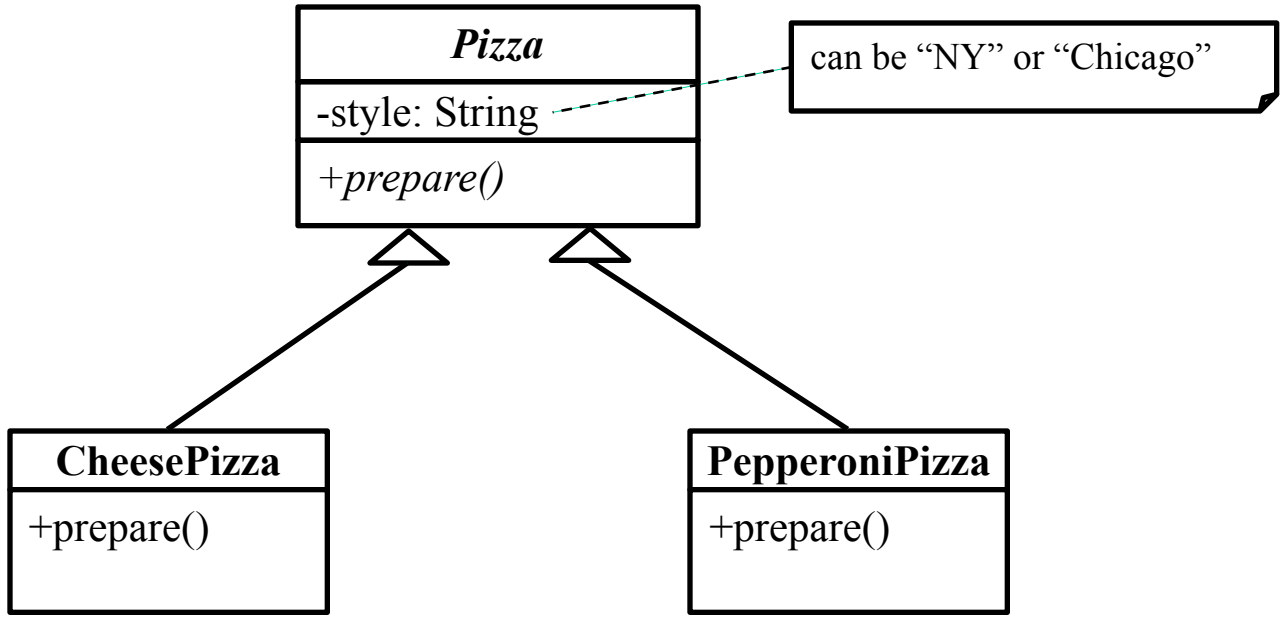
- In a pizza store system, two flavors of pizza are offered: Cheese Pizza and Pepperoni Pizza.





Requirements Statements$_2$

- Each flavor of pizza can be categorized into two styles: New York Style and Chicago Style.

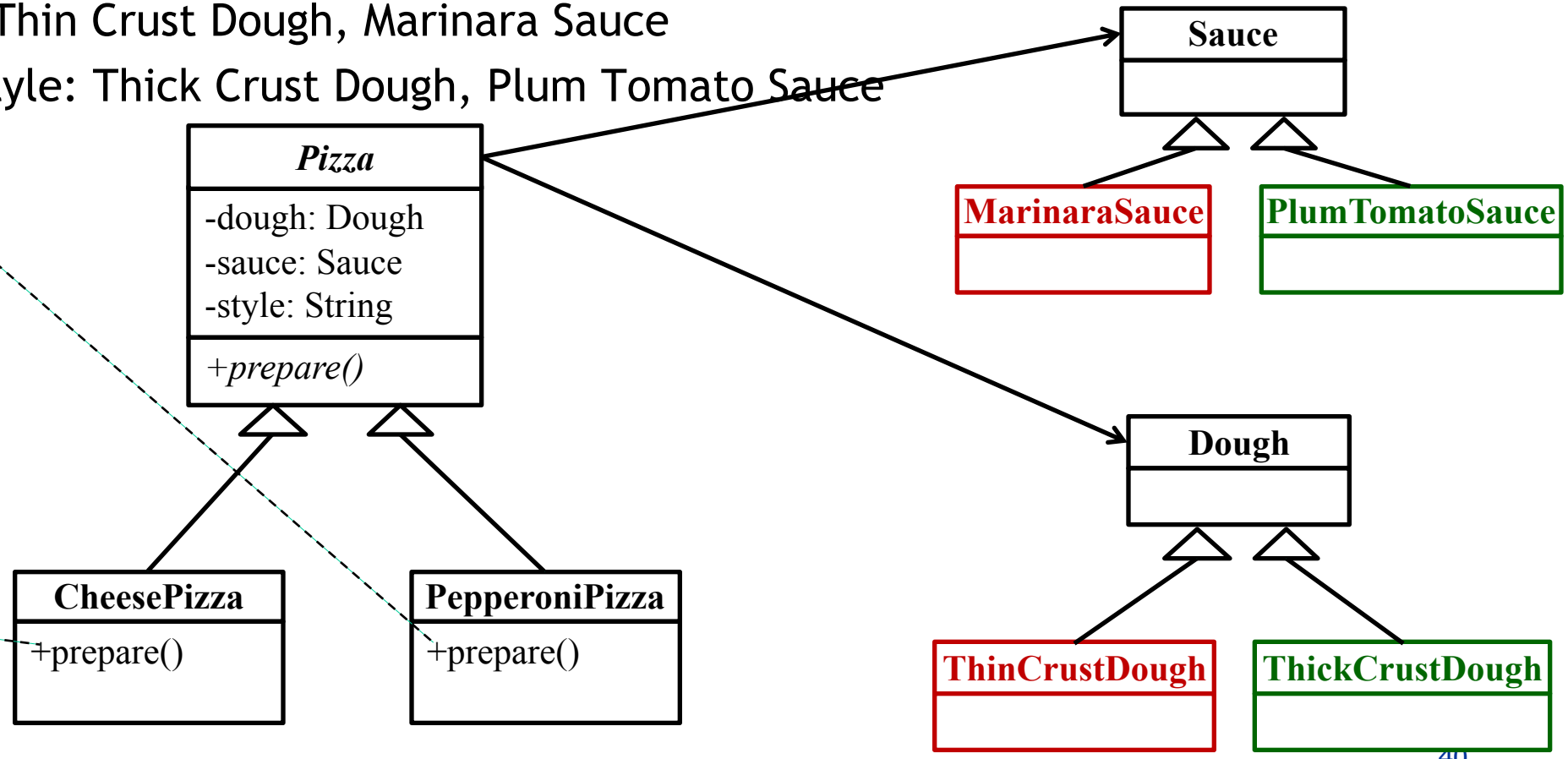


Requirements Statements$_3$

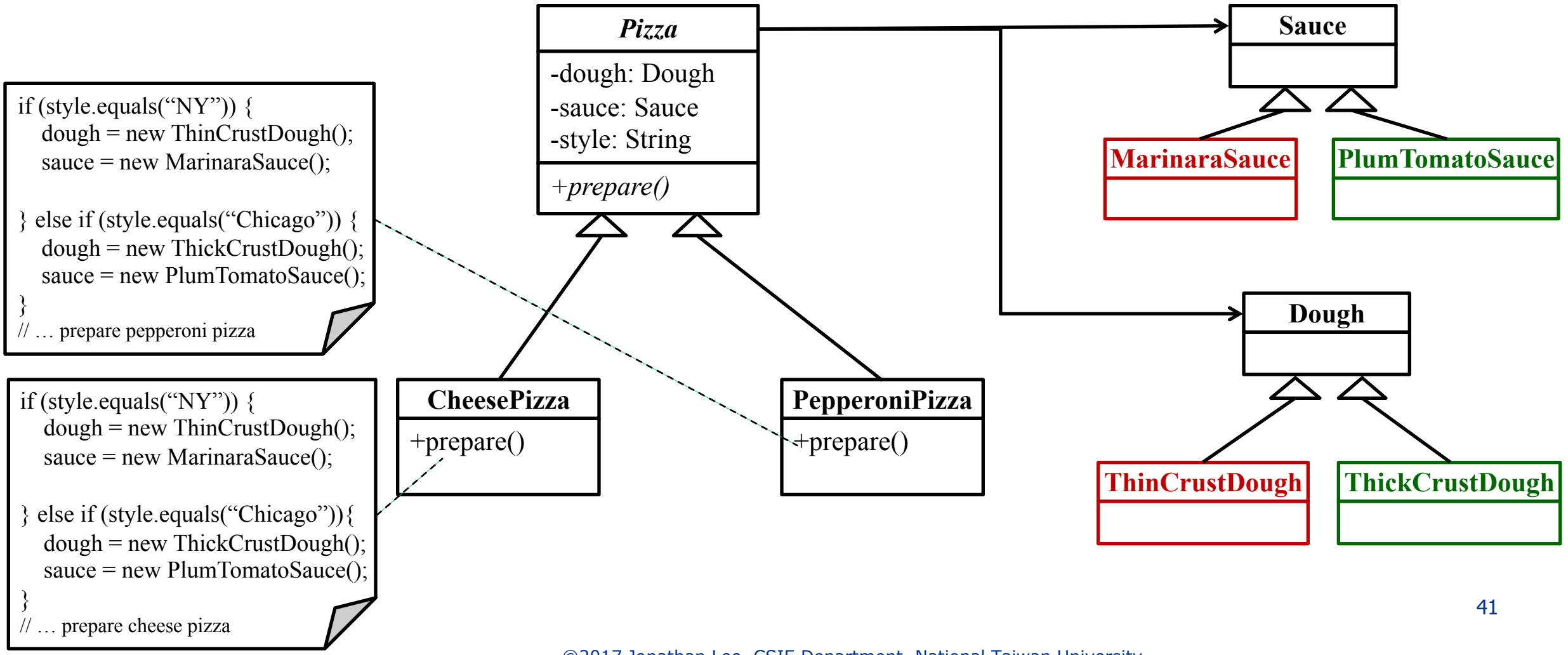
- Each pizza style requires its own type of dough and sauce, for example,
  - NY Style: Thin Crust Dough, Marinara Sauce
  - Chicago Style: Thick Crust Dough, Plum Tomato Sauce

```
if (style.equals("NY")) {
    dough = new ThinCrustDough();
    sauce = new MarinaraSauce();

} else if (style.equals("Chicago")) {
    dough = new ThickCrustDough();
    sauce = new PlumTomatoSauce();
}
// … prepare pepperoni pizza
```

```
if (style.equals("NY")) {
    dough = new ThinCrustDough();
    sauce = new MarinaraSauce();

} else if (style.equals("Chicago")){
    dough = new ThickCrustDough();
    sauce = new PlumTomatoSauce();
}
// … prepare cheese pizza
```



Initial Design - Class Diagram

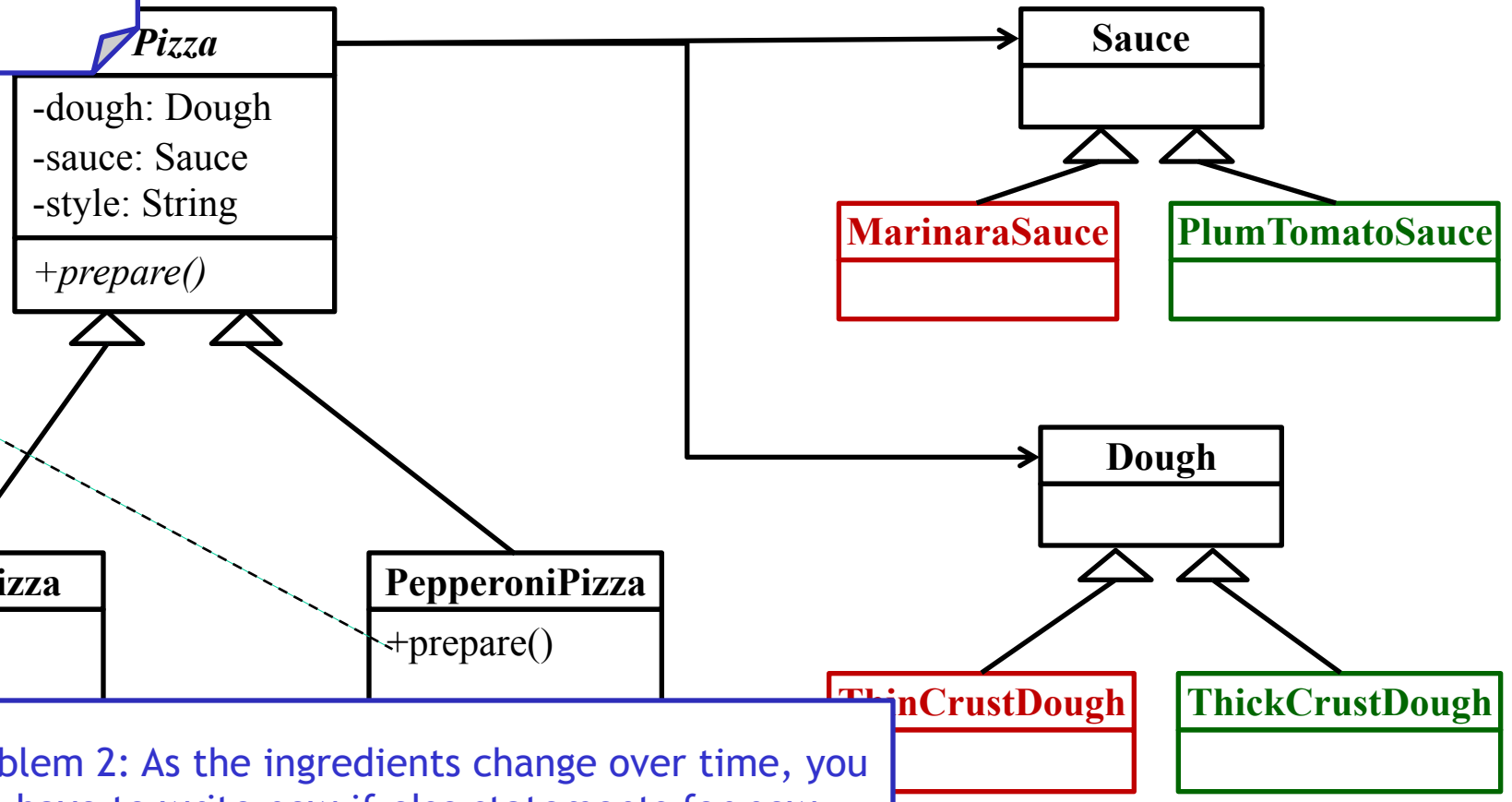


Problems with Initial Design

Problem 1: Duplicate code for preparing dough and sauce.

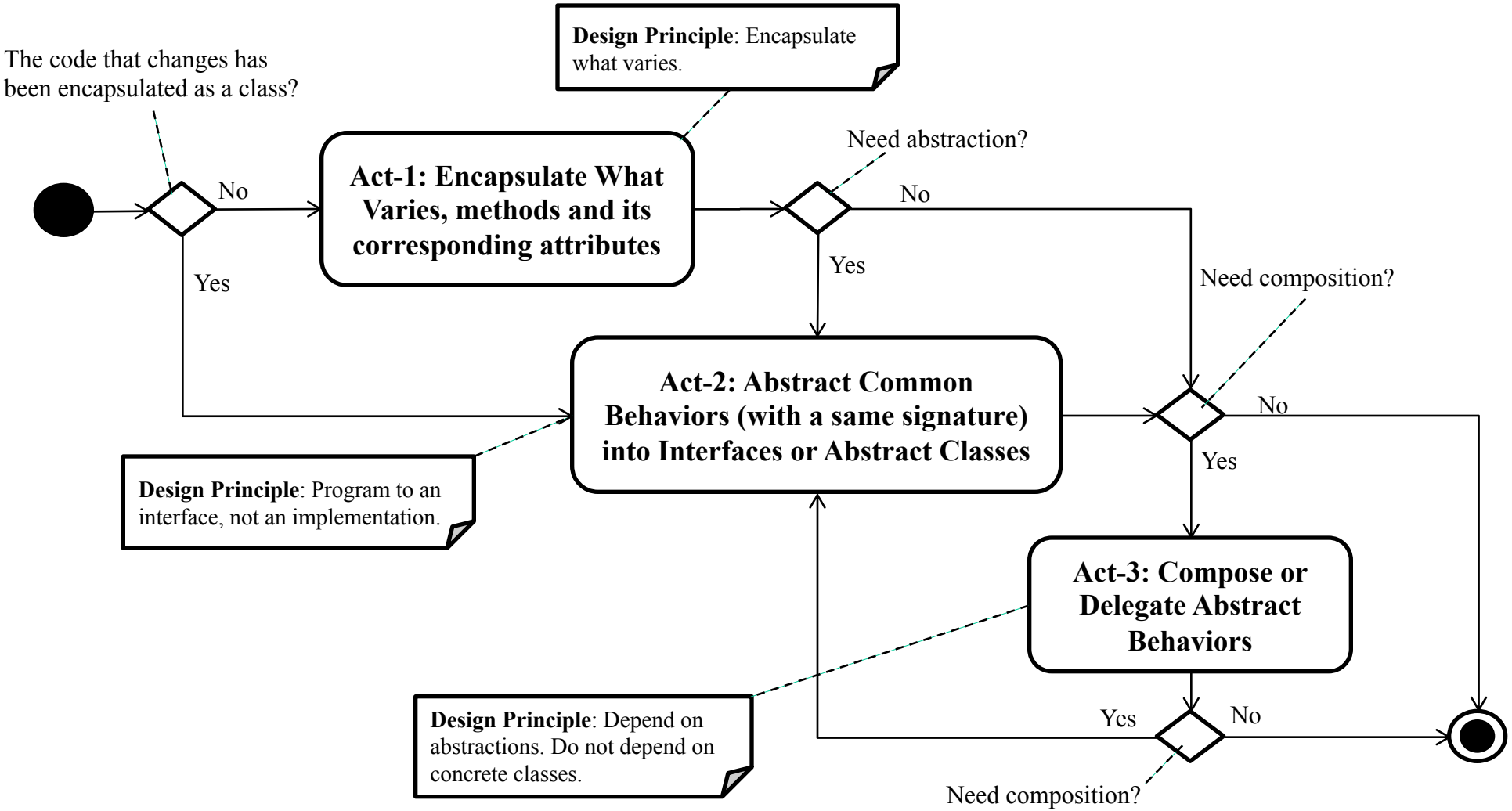
```
if (style.equals("NY")) {
    dough = new ThinCrustDough();
    sauce = new MarinaraSauce();

} else if (style.equals("Chicago")) {
    dough = new ThickCrustDough();
    sauce = new PlumTomatoSauce();
}
// … prepare pepperoni pizza
```

```
if (style.equals("NY")) {
    dough = new ThinCrustDough();
    sauce = new MarinaraSauce();

} else if (style.equals("Chicago")){
    dough = new ThickCrustDough();
    sauce = new PlumTomatoSauce();
}
// … prepare cheese pizza
```

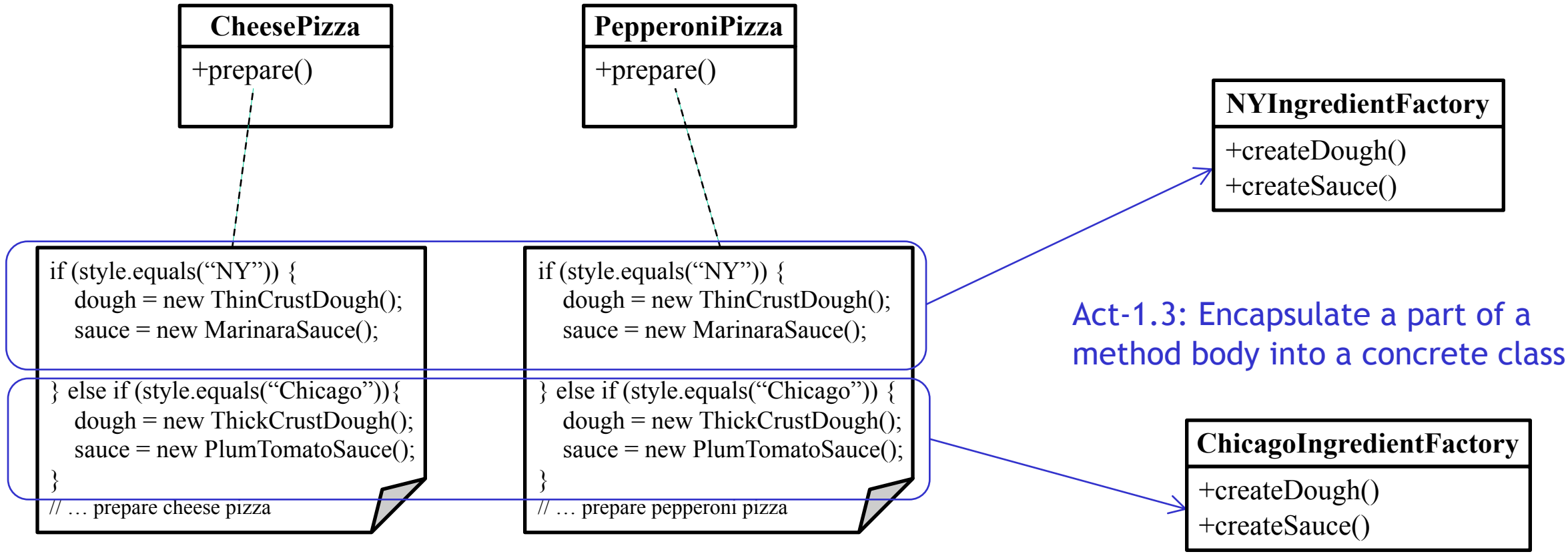


Problem 2: As the ingredients change over time, you will have to write new if-else statements for new pizza styles.

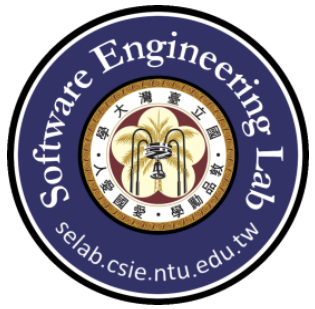
Design Process for Change



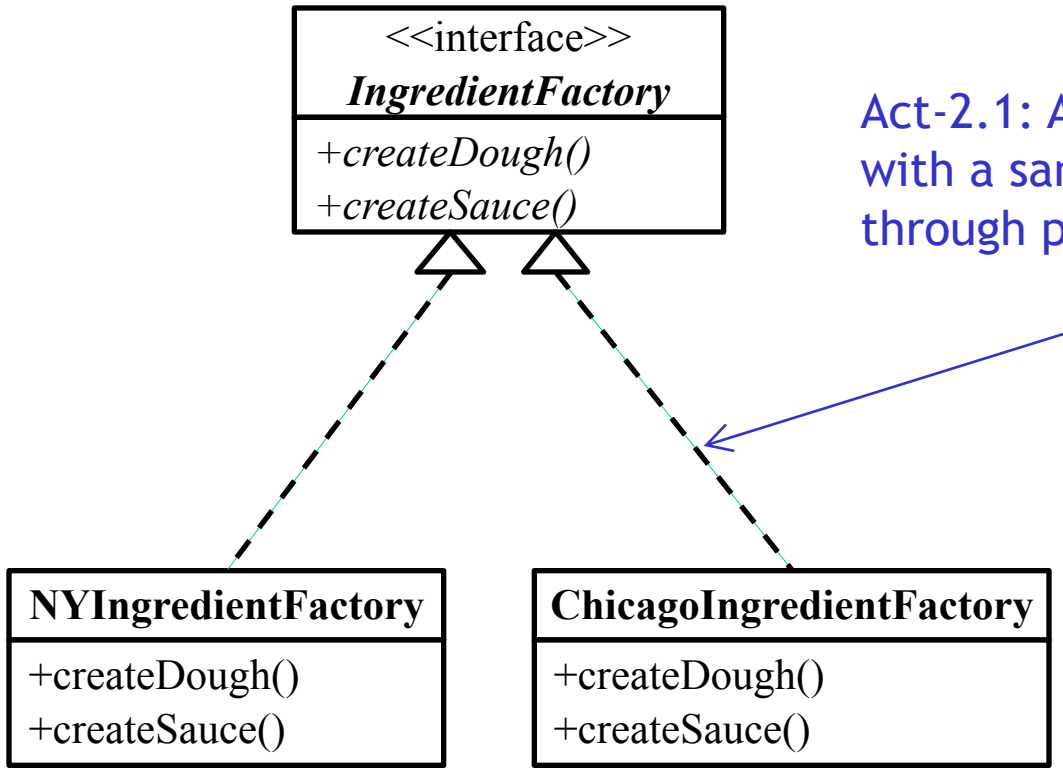
Act-1: Encapsulate What Varies



Act-1.3: Encapsulate a part of a method body into a concrete class

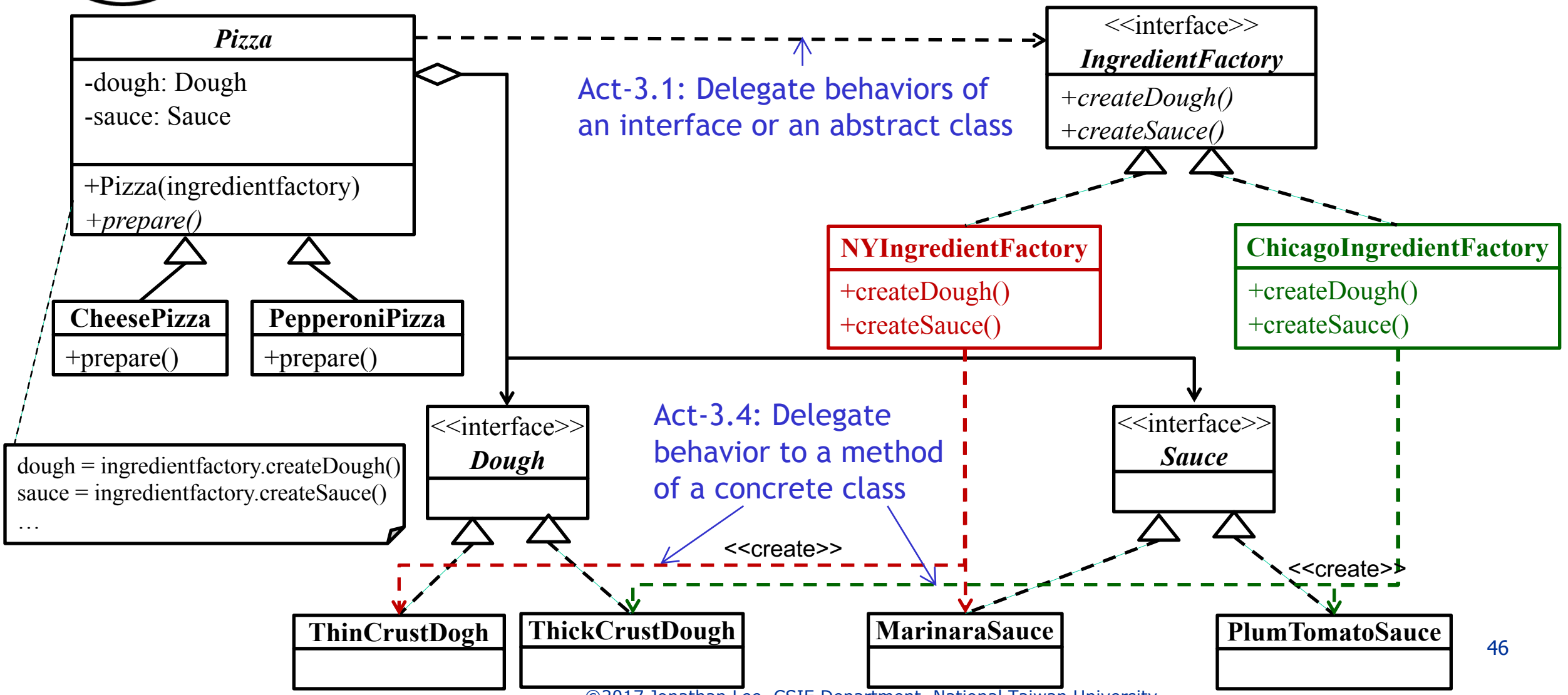


Act-2: Abstract Common Behaviors

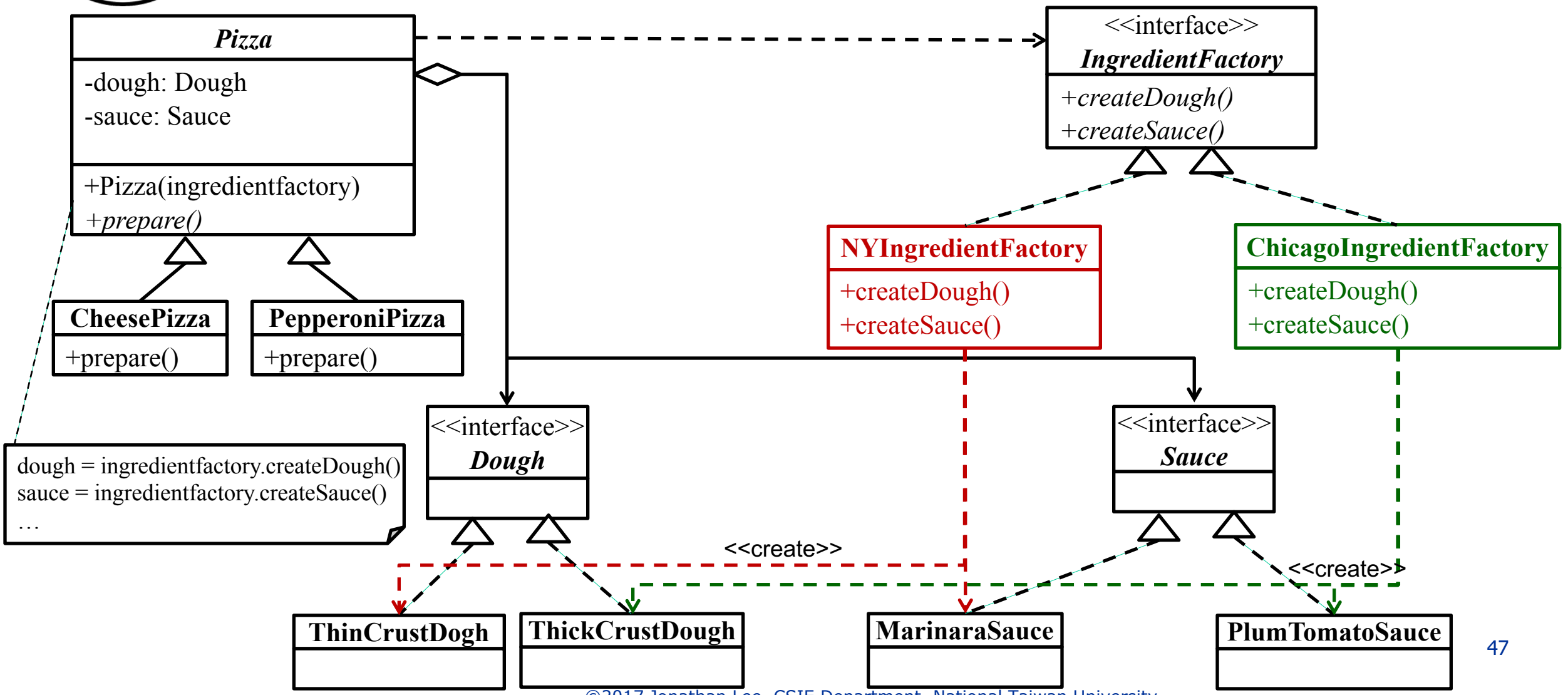


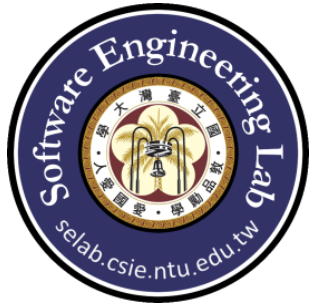
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

Act-3: Compose Abstract Behaviors



Refactored Design after Design Process

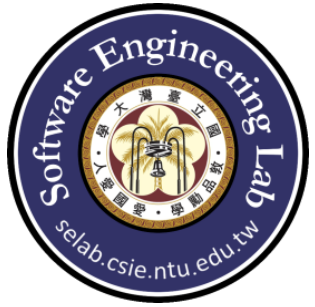




Source code

Pizza

```
public abstract class Pizza {
    private Dough dough;
    private Sauce sauce;

    public Pizza(IngredientFactory factory){
        this.dough = factory.createDough();
        this.sauce = factory.crateSauce();
        prepare();
    }
```



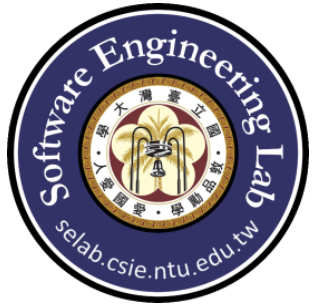
Source code

CheesePizza

```
public class CheesePizza extends Pizza{

    public CheesePizza(IngredientFactory factory) { super(factory); }

    @Override
    public void prepare() {
        System.out.println("Prepare Cheese Pizza with " + getDough().getClass().getName() + " and " + getSauce().getClass().getName());
    }

}
```

PepperoniPizza

```
public class PepperoniPizza extends Pizza{

    public PepperoniPizza(IngredientFactory factory) { super(factory); }

    @Override
    public void prepare() {
        System.out.println("Prepare Pepperoni Pizza with " + getDough().getClass().getName() + " and " + getSauce().getClass().getName());
    }

}
```



Source code

Dough

```
public interface Dough {

}
```

Sauce

```
public interface Sauce {

}
```

ThickCrustDough

```
public class ThickCrustDough implements Dough{

}
```

MarinaraSauce

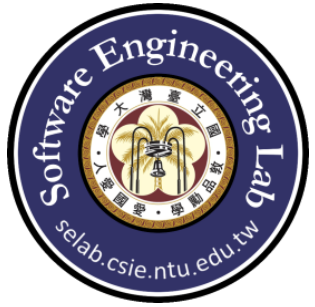
```
public class MarinaraSauce implements Sauce {

}
```

ThinCrustDough

```
public class ThinCrustDough implements Dough{

}
```

PlumtomatoSauce

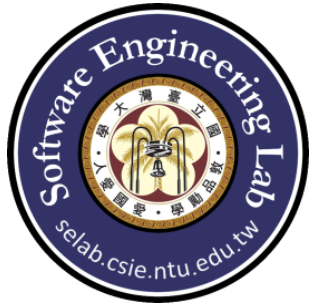
```
public class PlumTomatoSauce implements Sauce {

}
```



Source code

IngredientFactory

```
public interface IngredientFactory {

    public Dough createDough();

    public Sauce crateSauce();
}
```



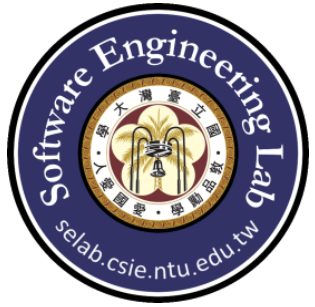
Source code

ChicagoIngredientFactory

```
public class ChicagoIngredientFactory implements IngredientFactory{

    @Override
    public Dough createDough() { return new ThickCrustDough(); }

    @Override
    public Sauce crateSauce() { return new PlumTomatoSauce(); }

}
```

NYIngredientFactory

```
public class NYIngredientFactory implements IngredientFactory{

    @Override
    public Dough createDough() { return new ThinCrustDough(); }

    @Override
    public Sauce crateSauce() { return new MarinaraSauce(); }

}
```



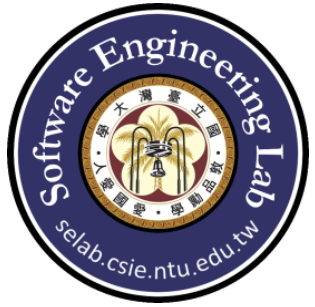
Input / Output

Input:

```
[flavor] [Pizza_style]

...
```

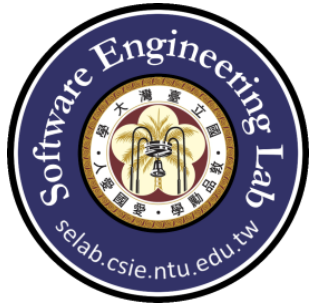
Output:

```
/*if [Pizza_style] is Chicago*/

Prepare [flavor] Pizza with ThickCrustDough and PlumTomatoSauce



/*if [Pizza_style] is NY*/

Prepare [flavor] Pizza with ThinCrustDough and MarinaraSauce



...
```



Test cases

- TestCase 1: Cheese NY Pizza
- TestCase 2: Pepperoni NY Pizza
- TestCase 3: Pepperoni Chicago Pizza
- TestCase 4: Cheese Chicago Pizza
- TestCase 5: Complex



Test cases

Test case1

1 Cheese NY	1 Prepare Cheese Pizza with ThinCrustDough and MarinaraSauce

Test case2

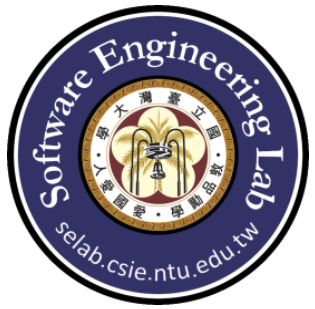
1 Pepperoni NY	1 Prepare Pepperoni Pizza with ThinCrustDough and MarinaraSauce

Test case3

1 Pepperoni Chicago	1 Prepare Pepperoni Pizza with ThickCrustDough and PlumTomatoSauce

Test case4

1 Cheese Chicago	1 Prepare Cheese Pizza with ThickCrustDough and PlumTomatoSauce



Test case5

Sample5.in

```
1  Cheese Chicago
2  Pepperoni Chicago
3  Pepperoni NY
4  Cheese NY
5  Cheese Chicago
6  Cheese NY
7  Pepperoni Chicago
8  Cheese Chicago
9  Pepperoni NY
```

Sample5.out

```
1  Prepare Cheese Pizza with ThickCrustDough and PlumTomatoSauce
2  Prepare Pepperoni Pizza with ThickCrustDough and PlumTomatoSauce
3  Prepare Pepperoni Pizza with ThinCrustDough and MarinaraSauce
4  Prepare Cheese Pizza with ThinCrustDough and MarinaraSauce
5  Prepare Cheese Pizza with ThickCrustDough and PlumTomatoSauce
6  Prepare Cheese Pizza with ThinCrustDough and MarinaraSauce
7  Prepare Pepperoni Pizza with ThickCrustDough and PlumTomatoSauce
8  Prepare Cheese Pizza with ThickCrustDough and PlumTomatoSauce
9  Prepare Pepperoni Pizza with ThinCrustDough and MarinaraSauce
```