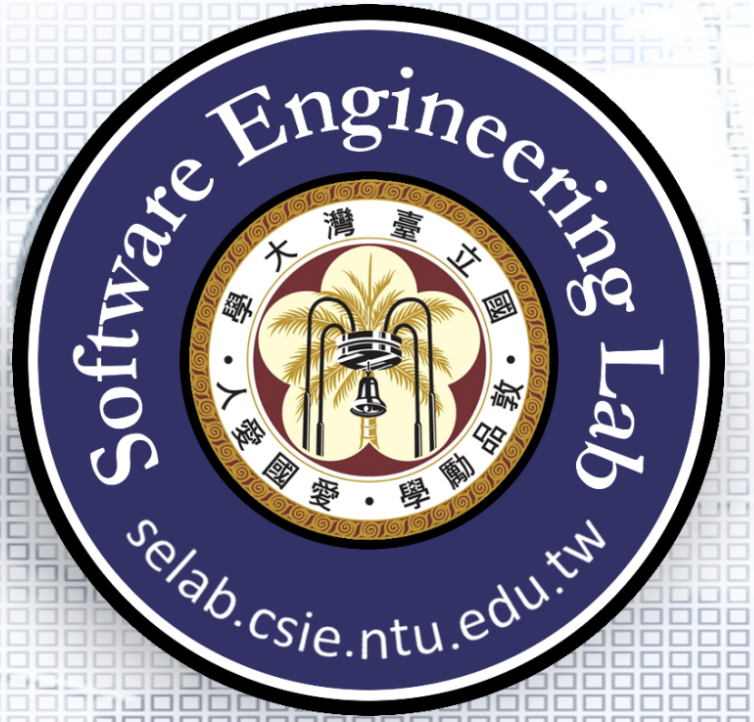
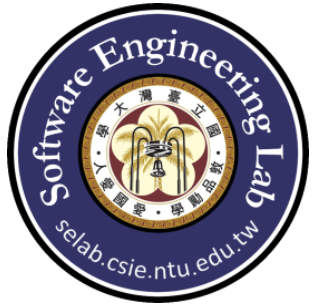


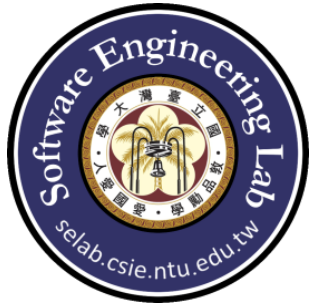
Factory Method Pattern

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



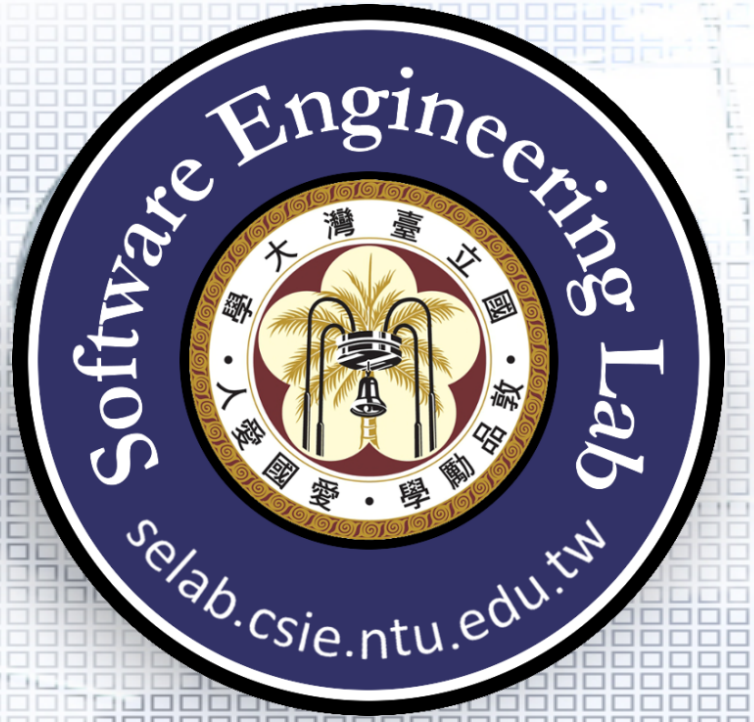
Design Aspect of Factory Method

Subclass of object that is instantiated



Outline

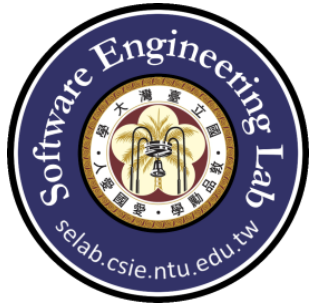
- Powerful Document Viewer Requirements Statements
- Initial Design and Its Problems
- Design Process
- Refactored Design after Design Process
- Recurrent Problems
- Intent
- Factory Method Pattern Structure
- Static Factory vs. Non-Static Factory
- Pizza Store: Another Example



Powerful Document Viewer (Factory Method)

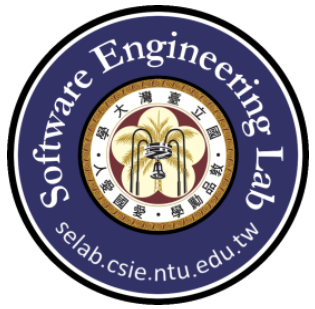
Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University



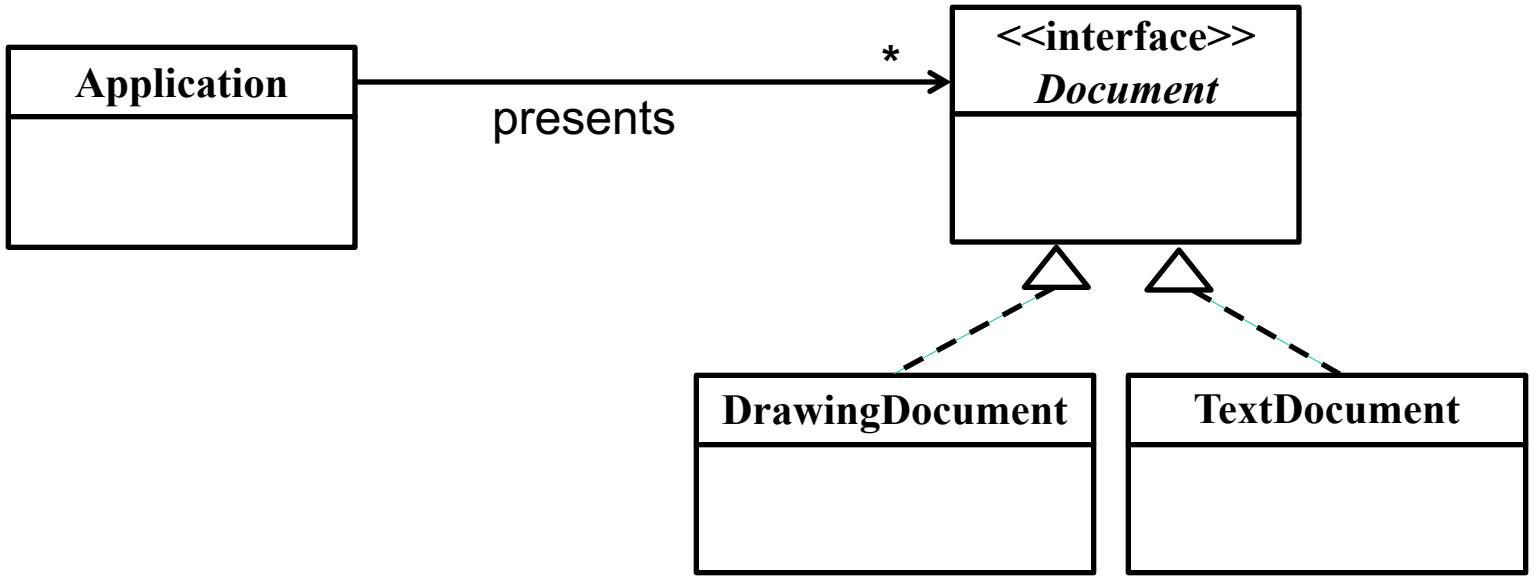
Requirements Statement

- A powerful application can present multiple documents at the same time.
- These documents include DrawingDocument, TextDocument, and so on.
- The application is responsible for managing documents and creating them as required.



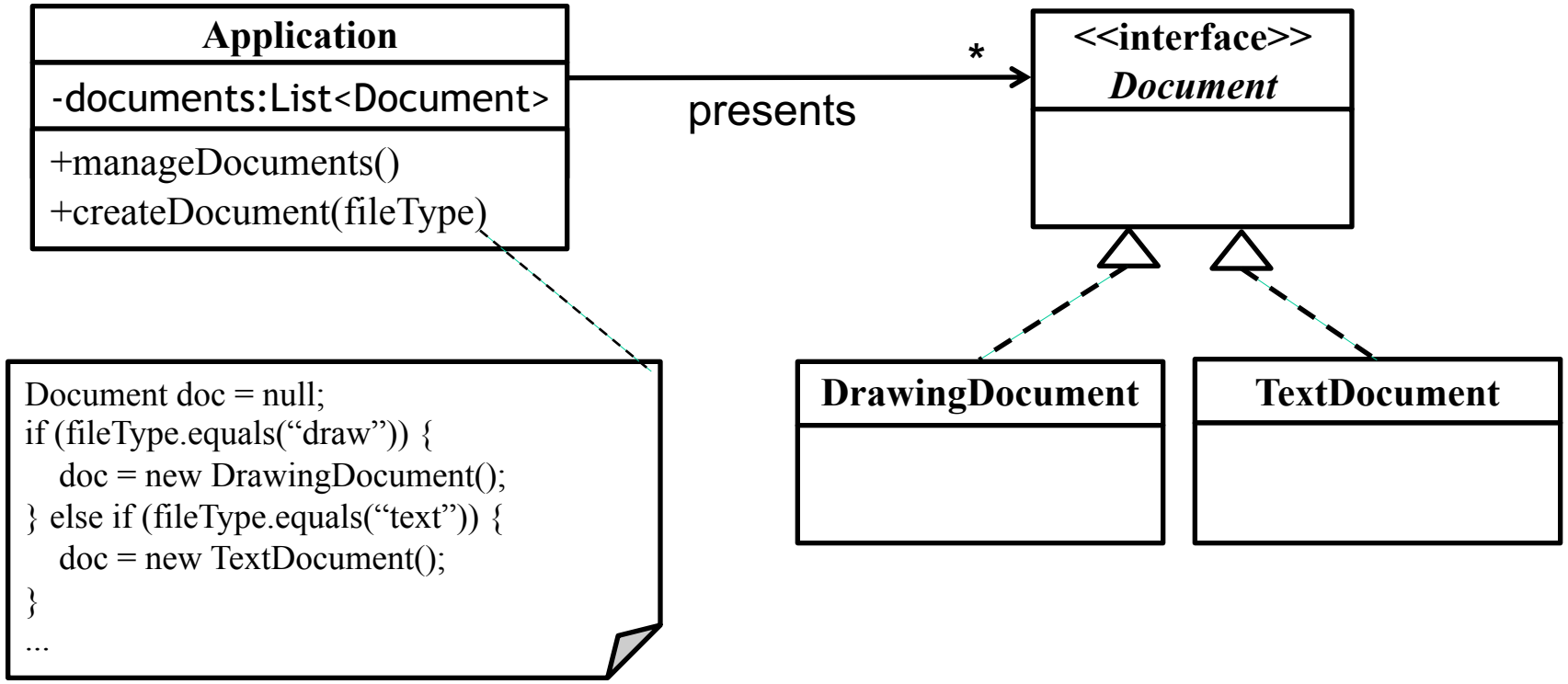
Requirements Statements$_1$

- A powerful application can present multiple documents at the same time.
- These documents include DrawingDocument, TextDocument, and so on.

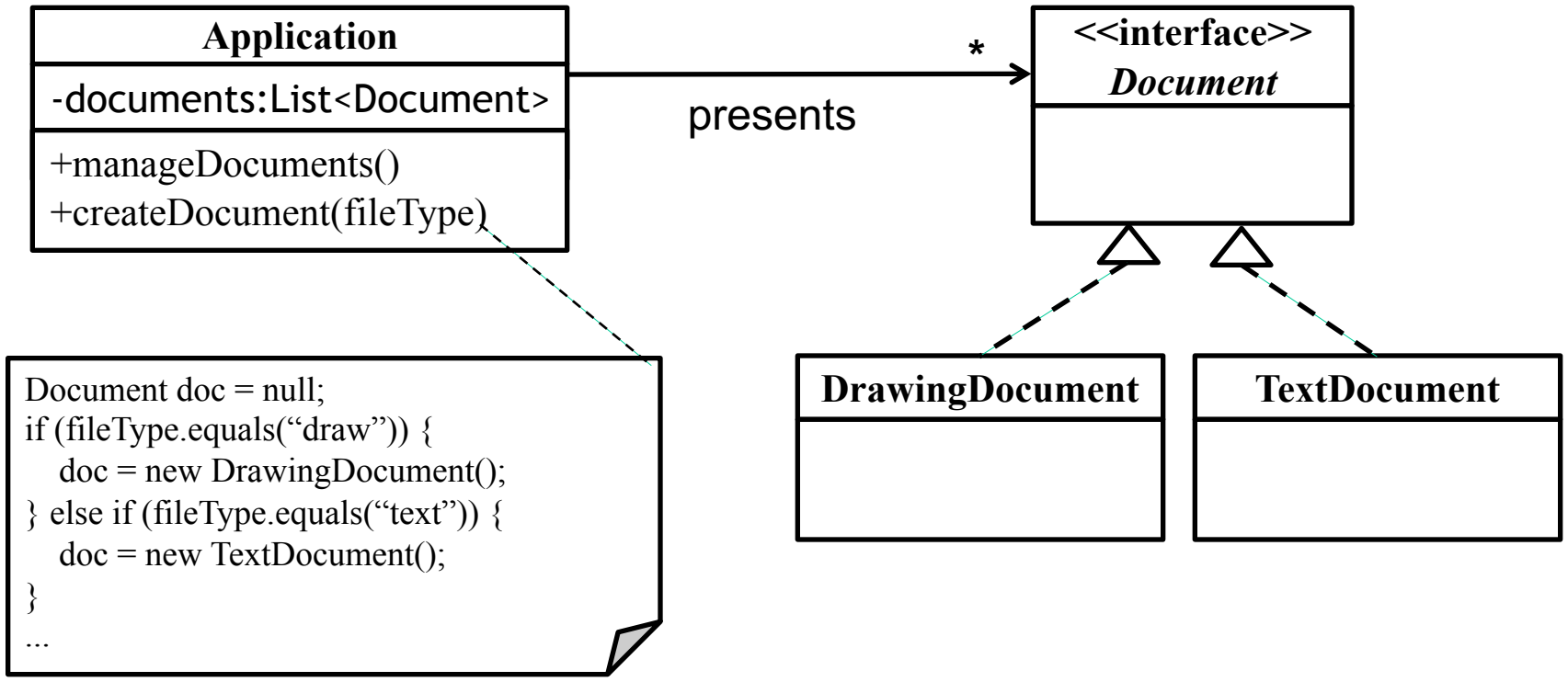


Requirements Statements$_2$

- The application is responsible for managing documents and will create them as required.

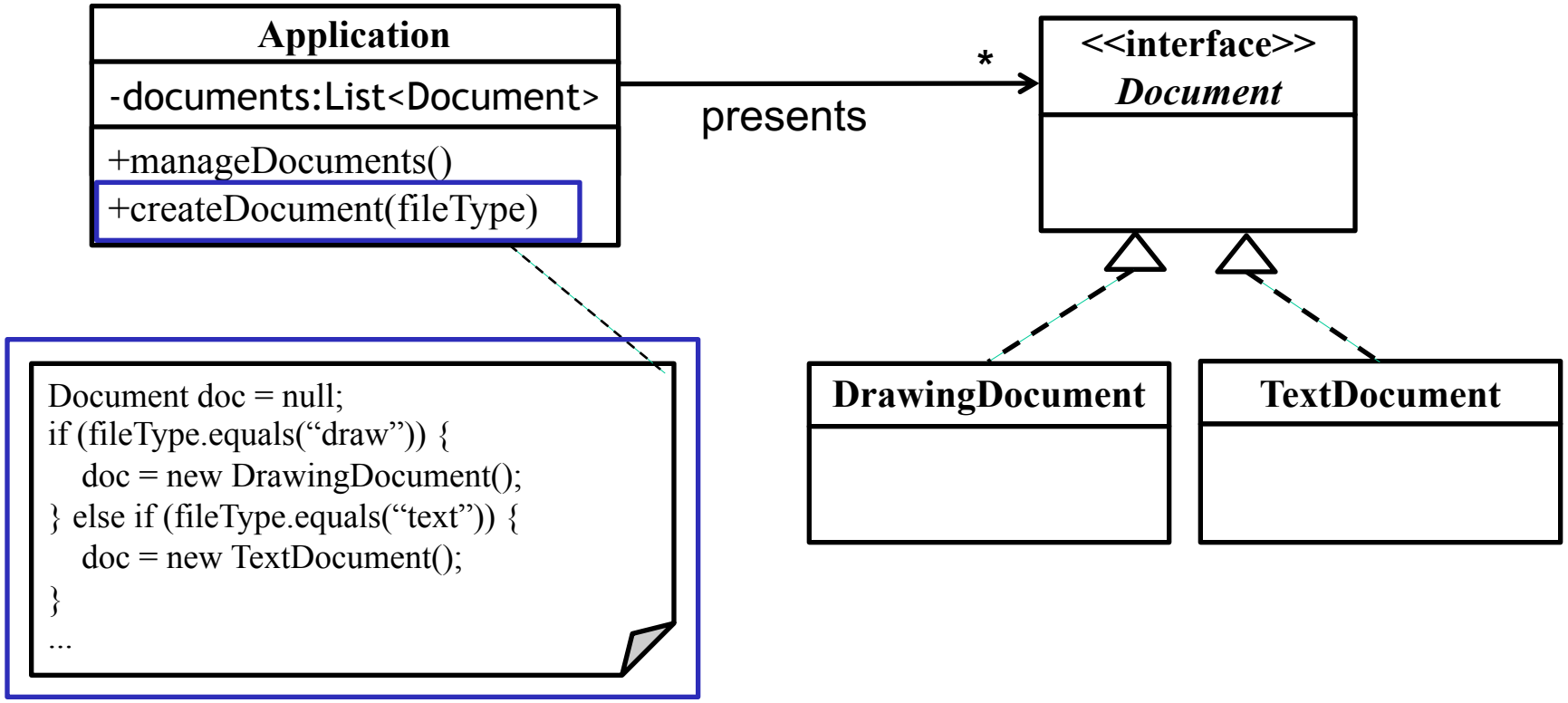


Initial Design

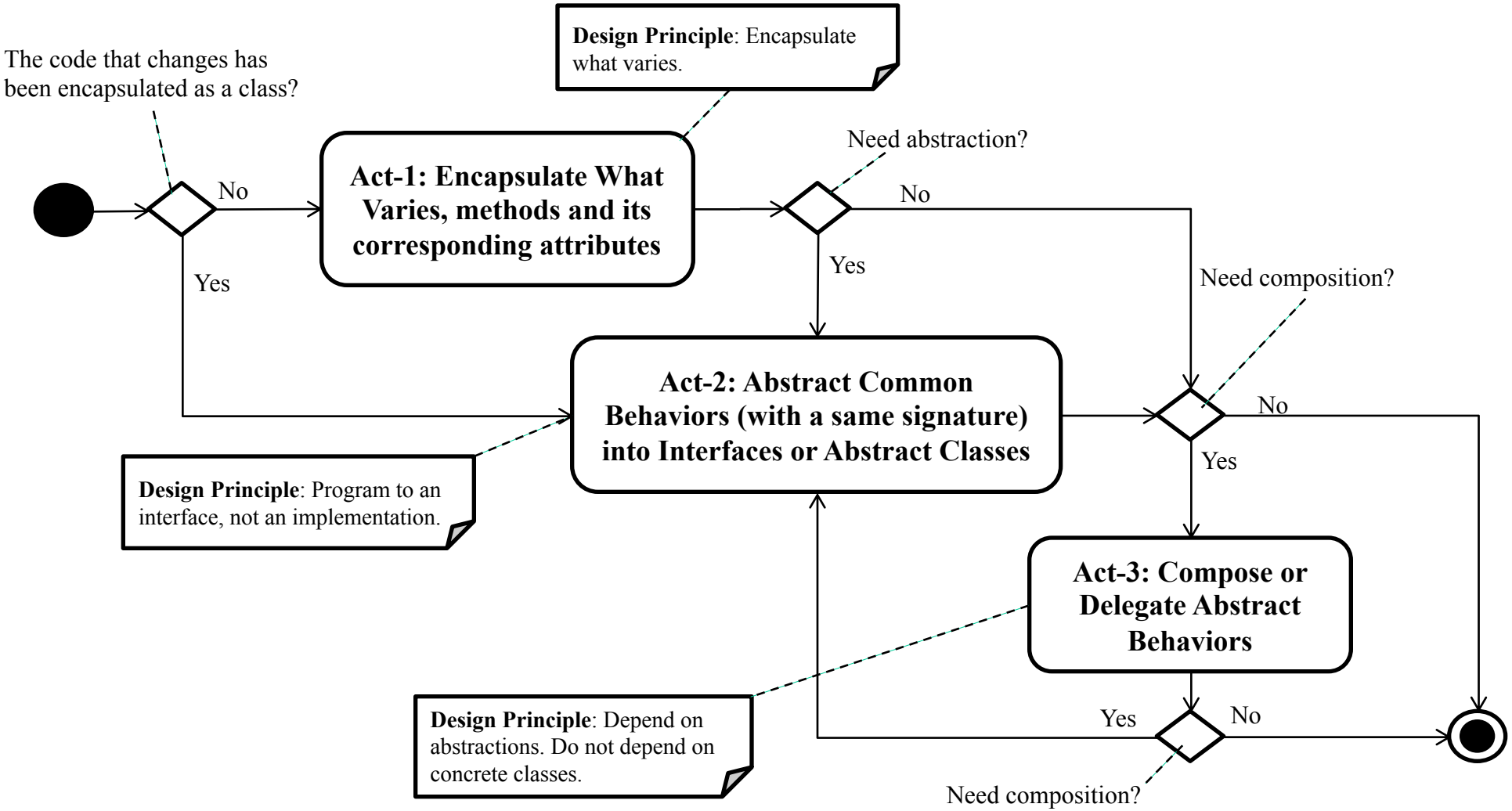


Problems with Initial Design

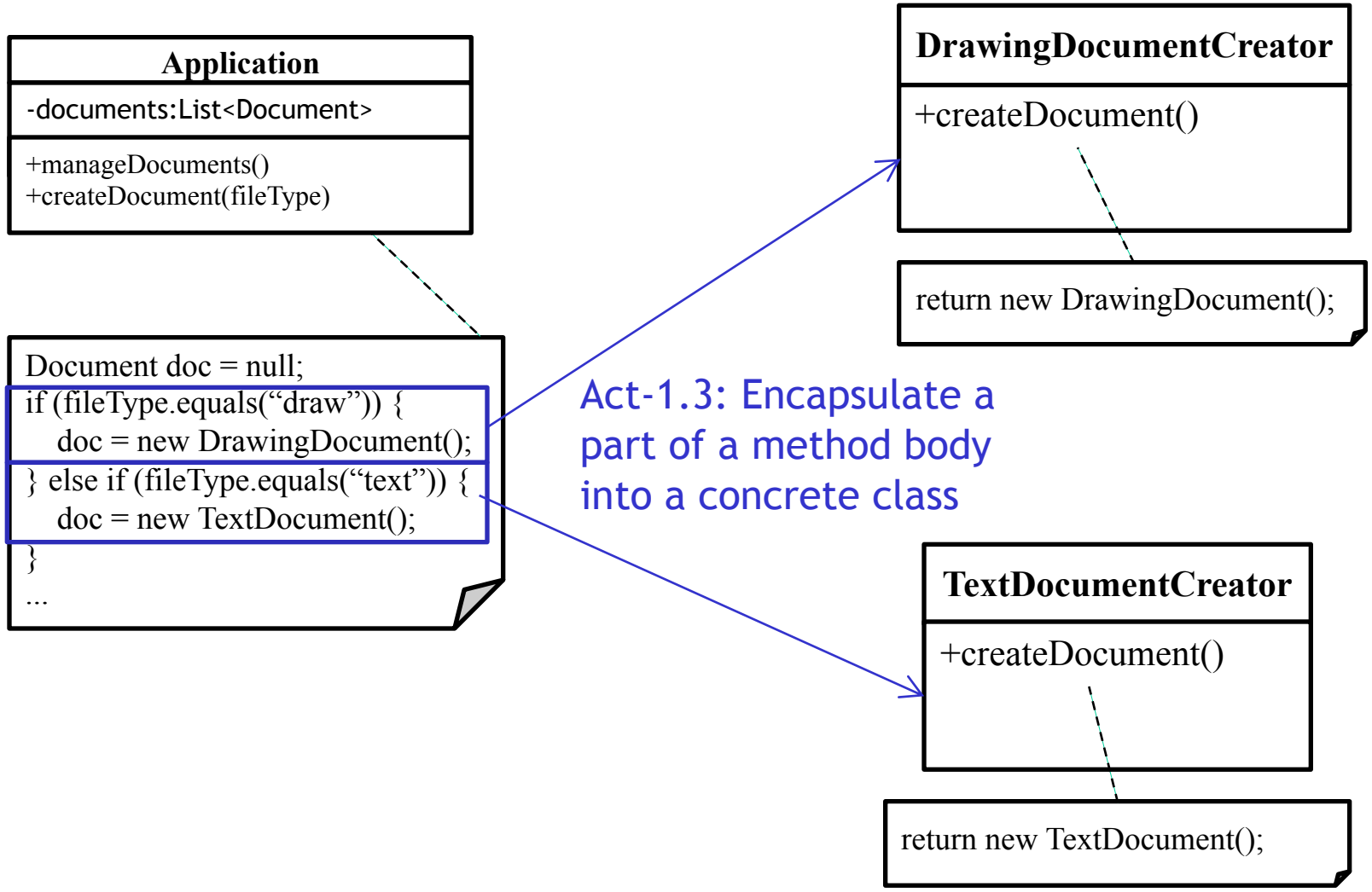
Problem: When more different kinds of documents are assigned to this application, we will have to extend the if-else statements for creation.



Design Process for Change

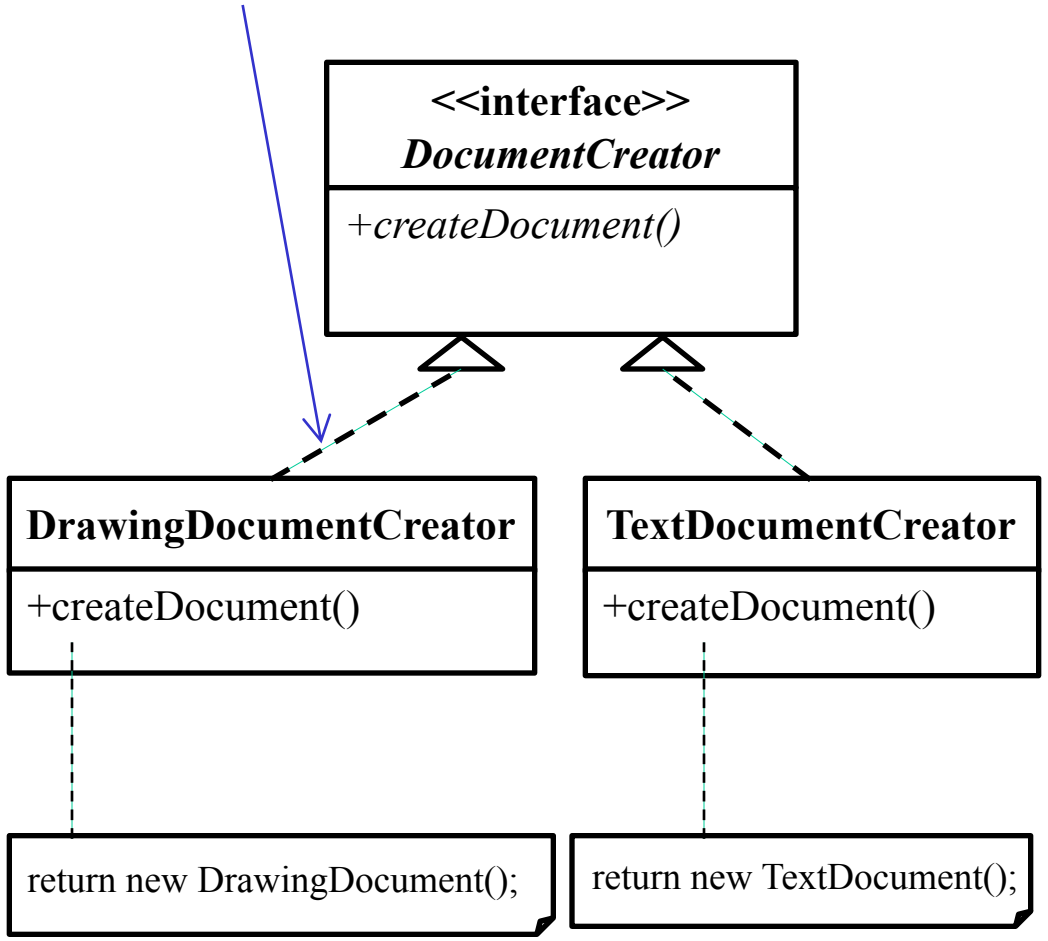


Act-1: Encapsulate What Varies

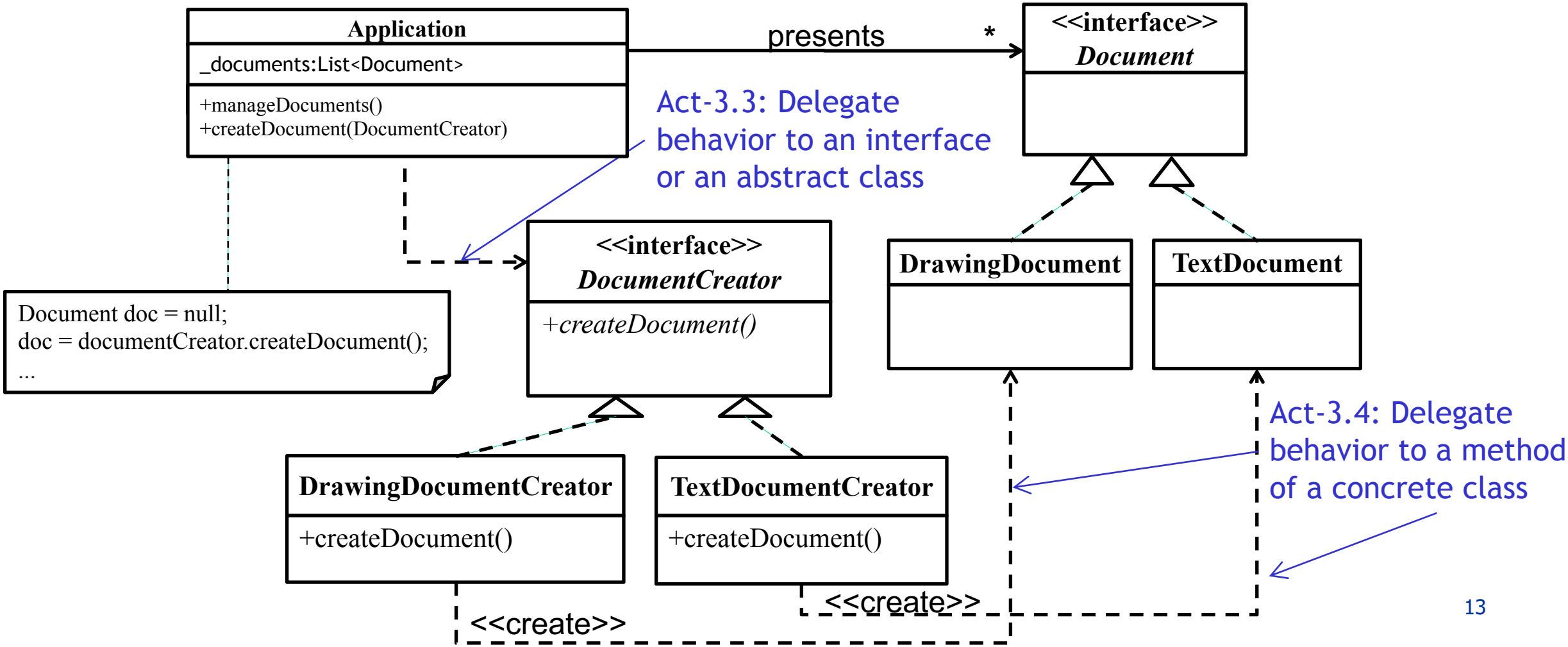


Act-2: Abstract Common Behaviors

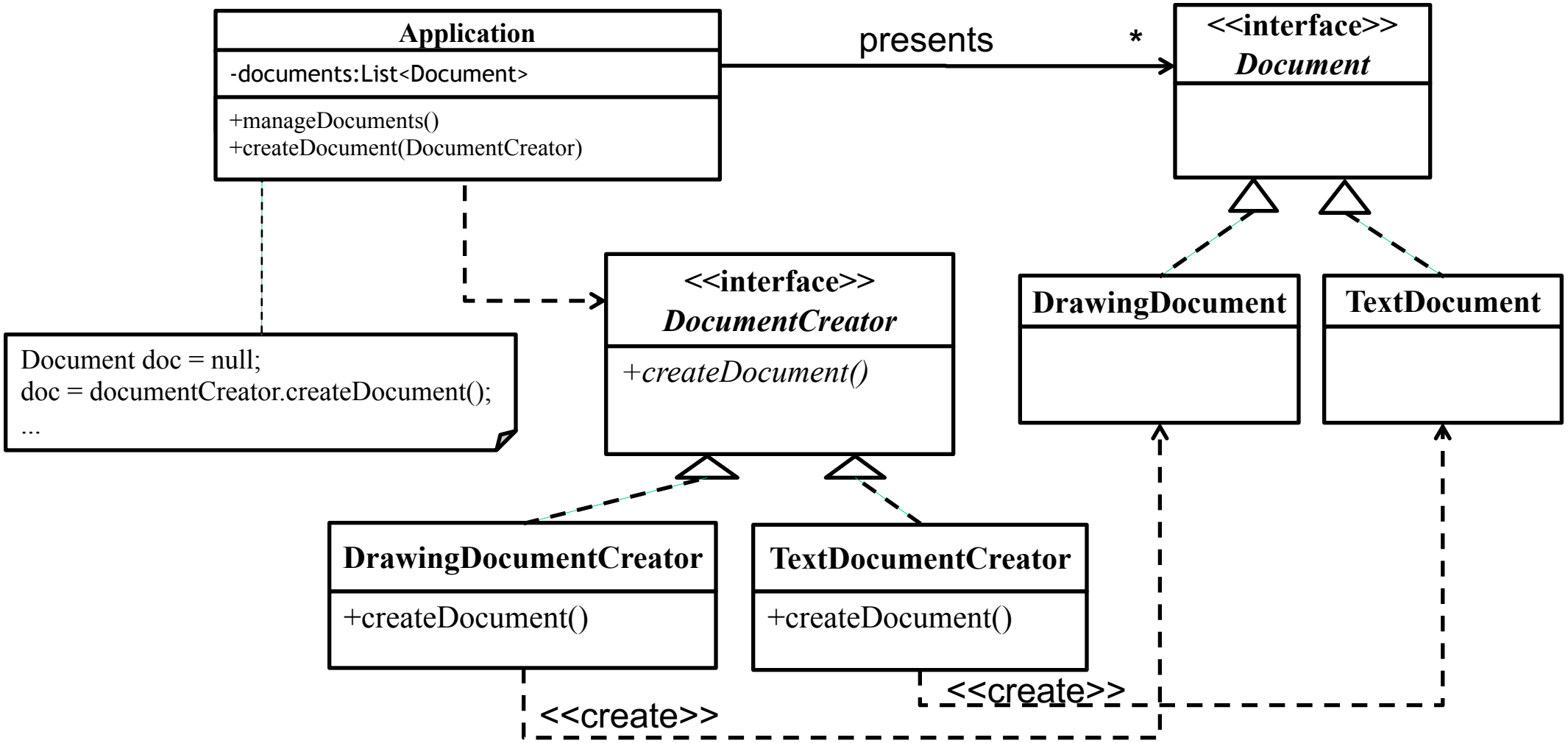
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism

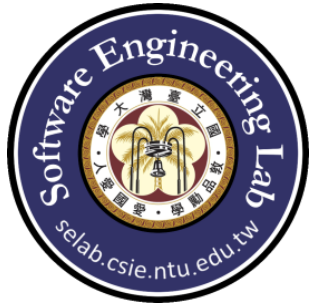


Act-3: Compose Abstract Behaviors



Refactored Design after Design Process

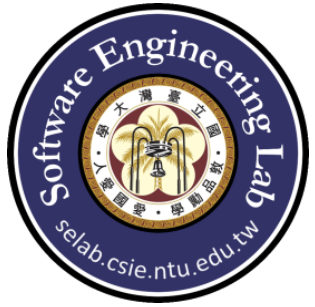




Source code

Application

```
public class Application {
    private List<Document> documents = new ArrayList<>();

    public void createDocument(DocumentCreator documentCreator){
        documents.add(documentCreator.createDocument());
    }


    public void present(){
        for(Document document: documents){
            System.out.println(document.getClass().getName());
        }
    }


}
```



Source code

DocumentCreator

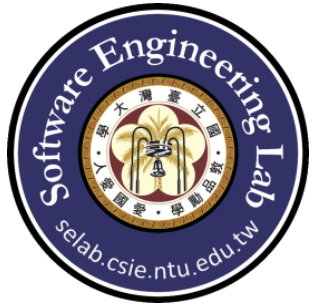
```
public interface DocumentCreator {

    public Document createDocument();
}
```

TextDocumentCreator

```
public class TextDocumentCreator implements DocumentCreator {
    @Override
    public Document createDocument() {
        return new TextDocument();
    }
}
```

DrawingDocumentCreator

```
public class DrawingDocumentCreator implements DocumentCreator {
    @Override
    public Document createDocument() { return new DrawingDocument(); }

}
```



Source code

Document

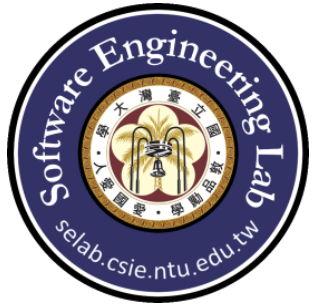
```
public interface Document {

}
```

DrawingDocument

```
public class DrawingDocument implements Document{

}
```

TextDocument

```
public class TextDocument implements Document{

}
```



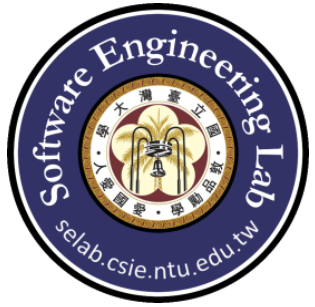
Input/Output

Input:

```
[Create_document_type]

Present /*extra command, show all documents to standard output
with the sequential order from input*/

...
```

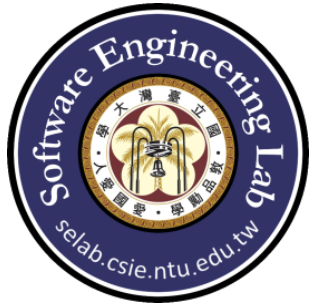
Output:

```
 [Type_Document]

...
```



Test cases

- TestCase 1: only Text
- TestCase 2: only Draw
- TestCase 3: both Text and Draw
- TestCase 4: Complex



Test cases

Test cases1

Sample1.in

```
1 Text
2 Present
```

Sample1.out

```
1 TextDocument
2
```

Test cases2

Sample2.in

```
1 Draw
2 Present
```

Sample2.out

```
1 DrawingDocument
2
```

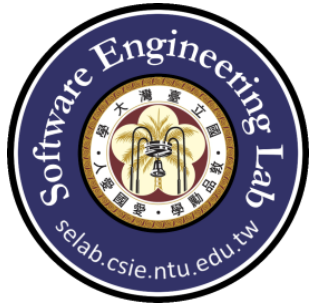
Test cases3

Sample3.in

```
1 Draw
2 Text
3 Present
```

Sample3.out

```
1 DrawingDocument
2 TextDocument
3
```



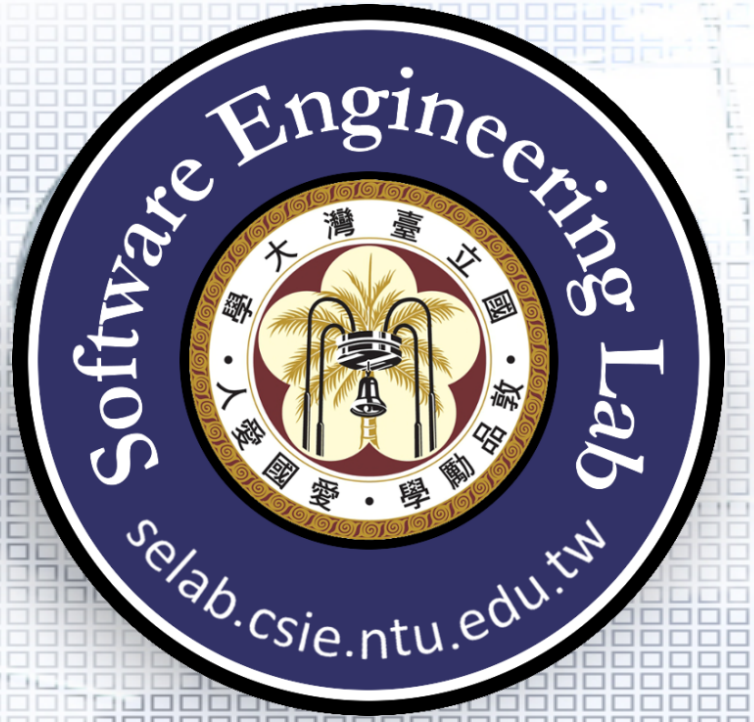
Test case4

Sample4.in

```
1  Draw
2  Text
3  Present
4  Text
5  Present
6  Draw
7  Draw
8  Draw
9  Draw
10 Present
11 Text
12 Present
13 Draw
14 Present
```

Sample4.out

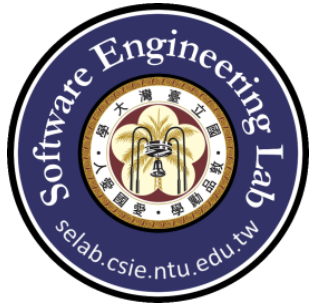
```
1  DrawingDocument
2  TextDocument
3  DrawingDocument
4  TextDocument
5  TextDocument
6  DrawingDocument
7  TextDocument
8  TextDocument
9  DrawingDocument
10 DrawingDocument
11 DrawingDocument
12 DrawingDocument
13 DrawingDocument
14 TextDocument
15 TextDocument
16 DrawingDocument
17 DrawingDocument
18 DrawingDocument
19 DrawingDocument
20 TextDocument
21 DrawingDocument
22 TextDocument
23 TextDocument
24 DrawingDocument
25 DrawingDocument
26 DrawingDocument
27 DrawingDocument
28 TextDocument
29 DrawingDocument
```



Pizza Store (Factory Method)

Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University

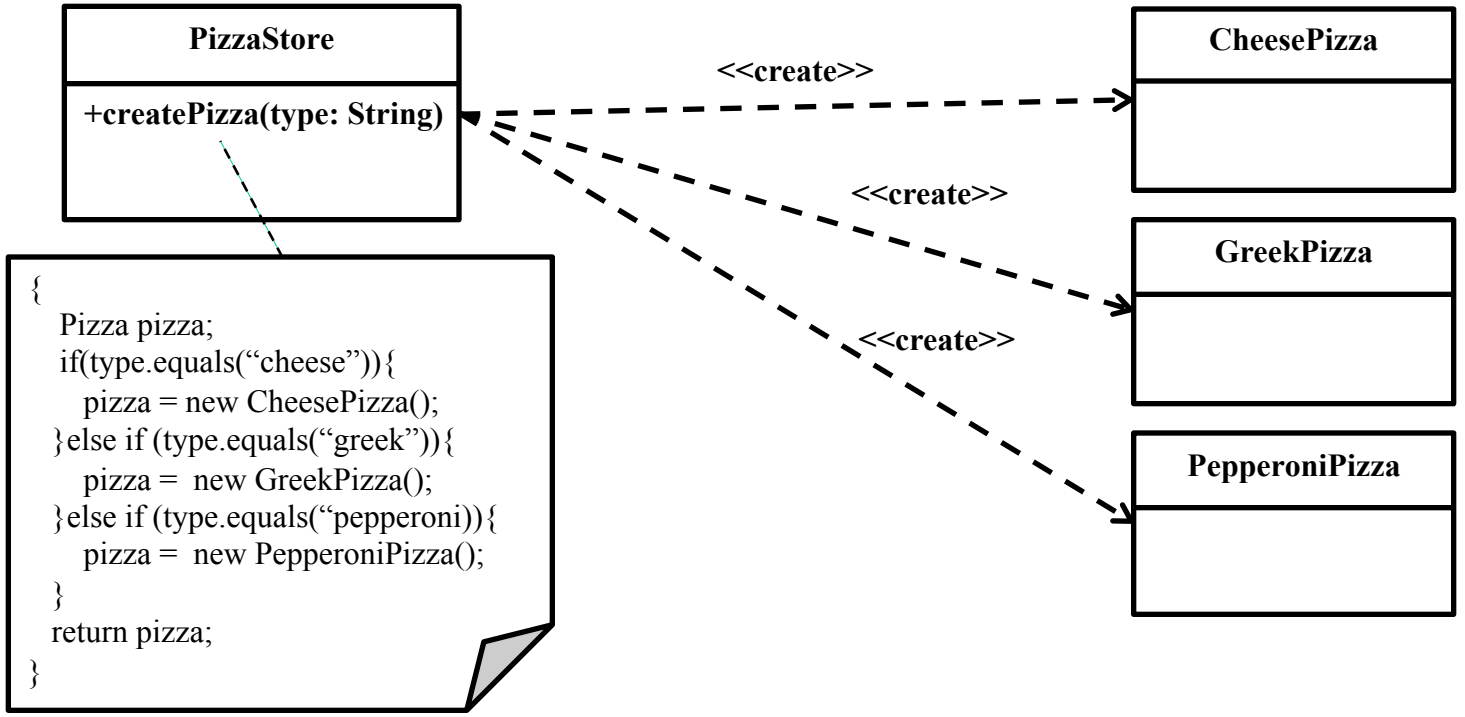


Requirements Statement

- Pizza Store makes more than one type of pizza: Cheese Pizza, Greek Pizza, and Pepperoni Pizza.
- Each pizza has different way to prepare, and has the same way to bake, to cut, and to box.
- To make this store more competitive, you may add a new flavor of pizza or remove unpopular ones.

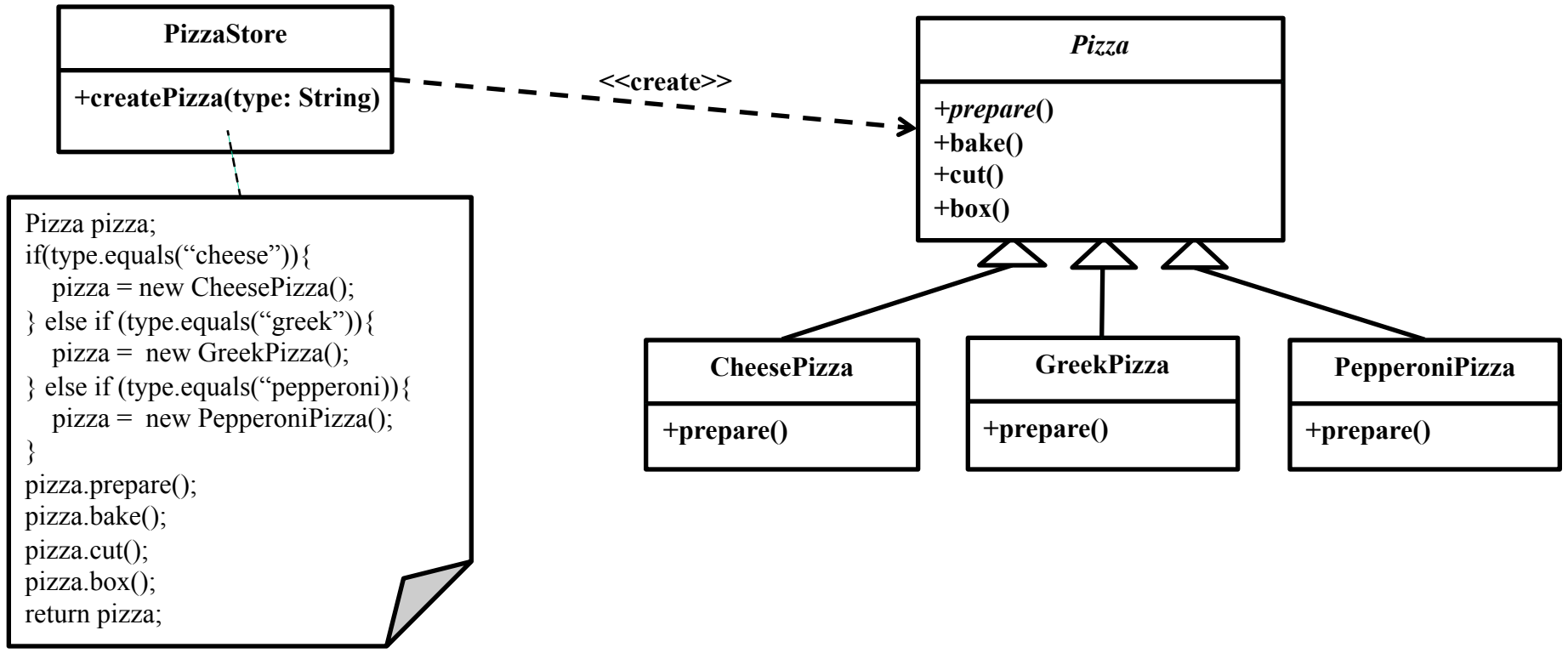
Requirements Statements$_1$

- Pizza store makes more than one type of pizza: Cheese Pizza, Greek Pizza, and Pepperoni Pizza.



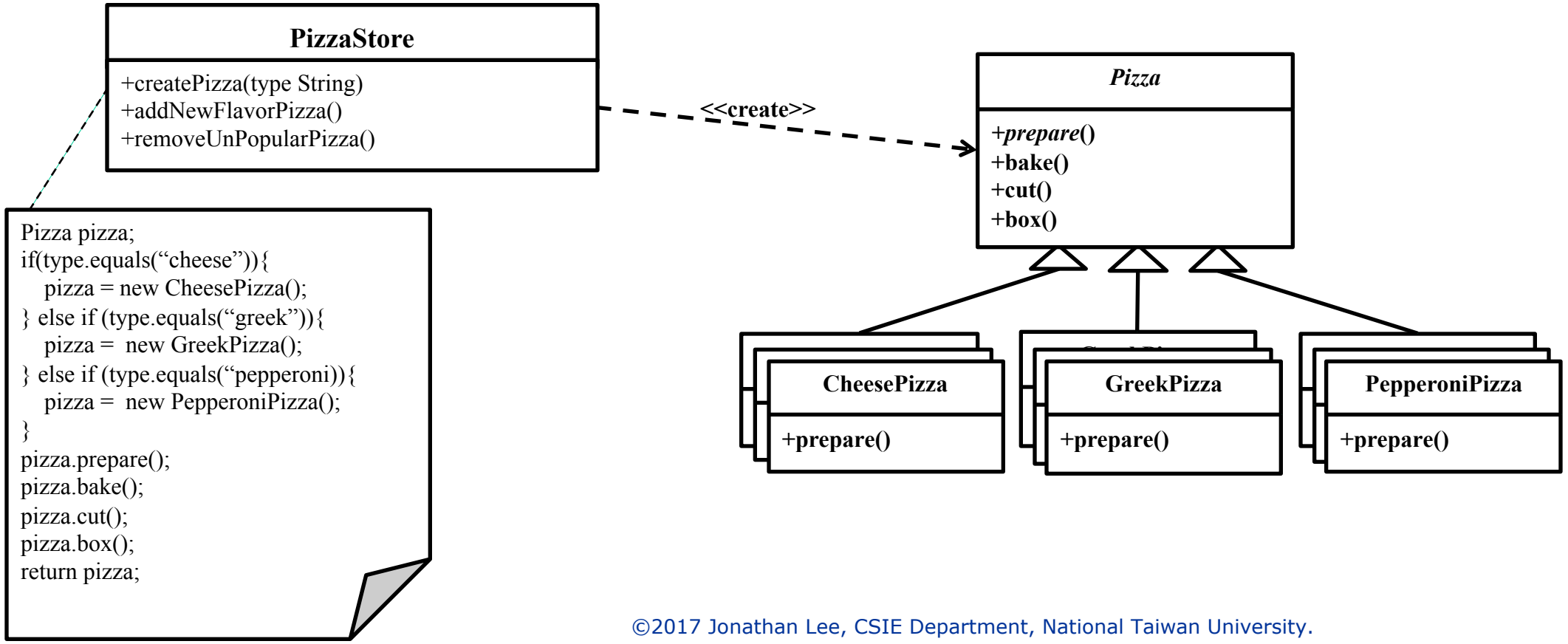
Requirements Statements$_2$

- Each pizza has different way to prepare, and has the same way to bake, to cut, and to box.

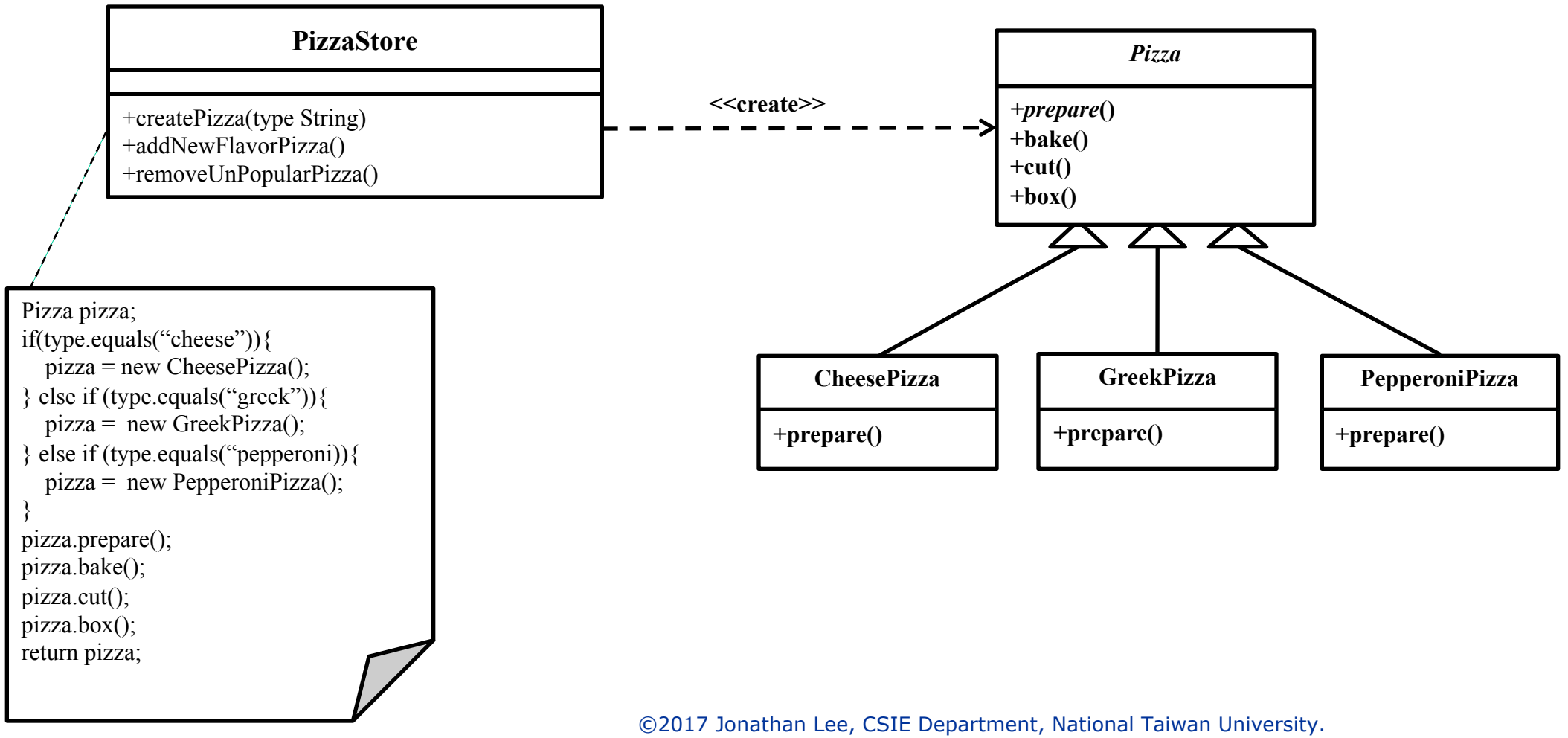


Requirements Statements$_3$

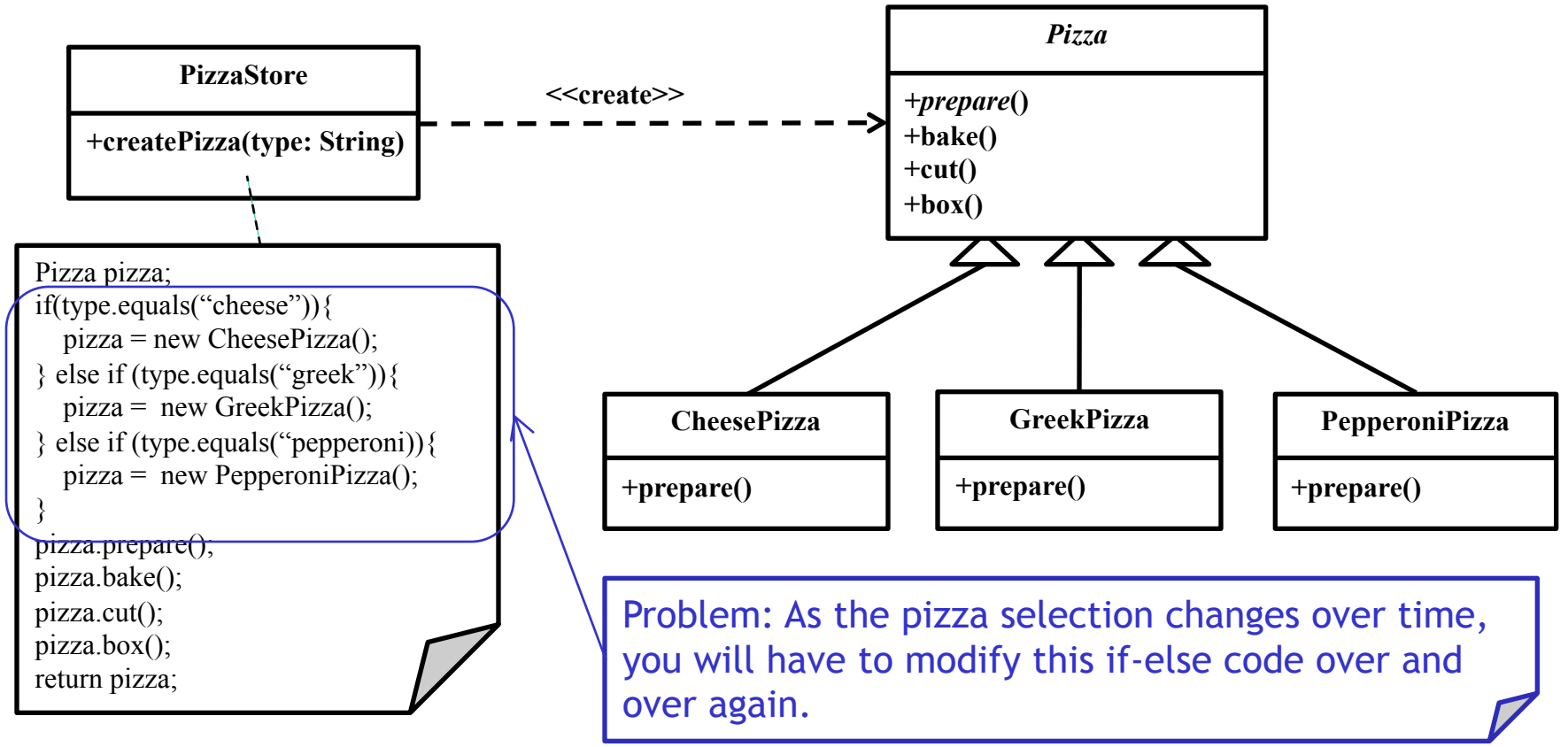
- To make this store more competitive, you may add a new flavor of pizza or remove unpopular ones.



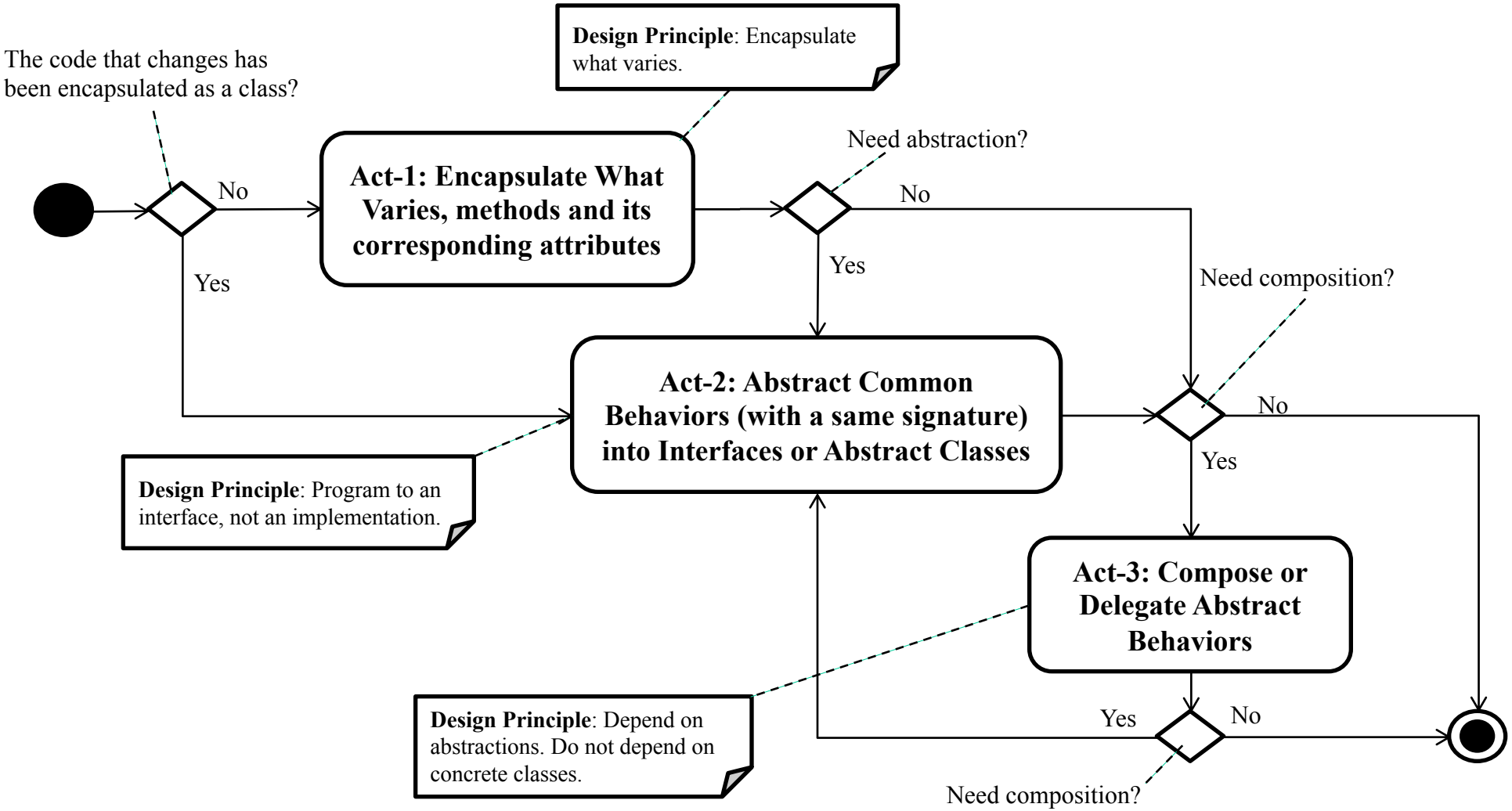
Initial Design - Class Diagram



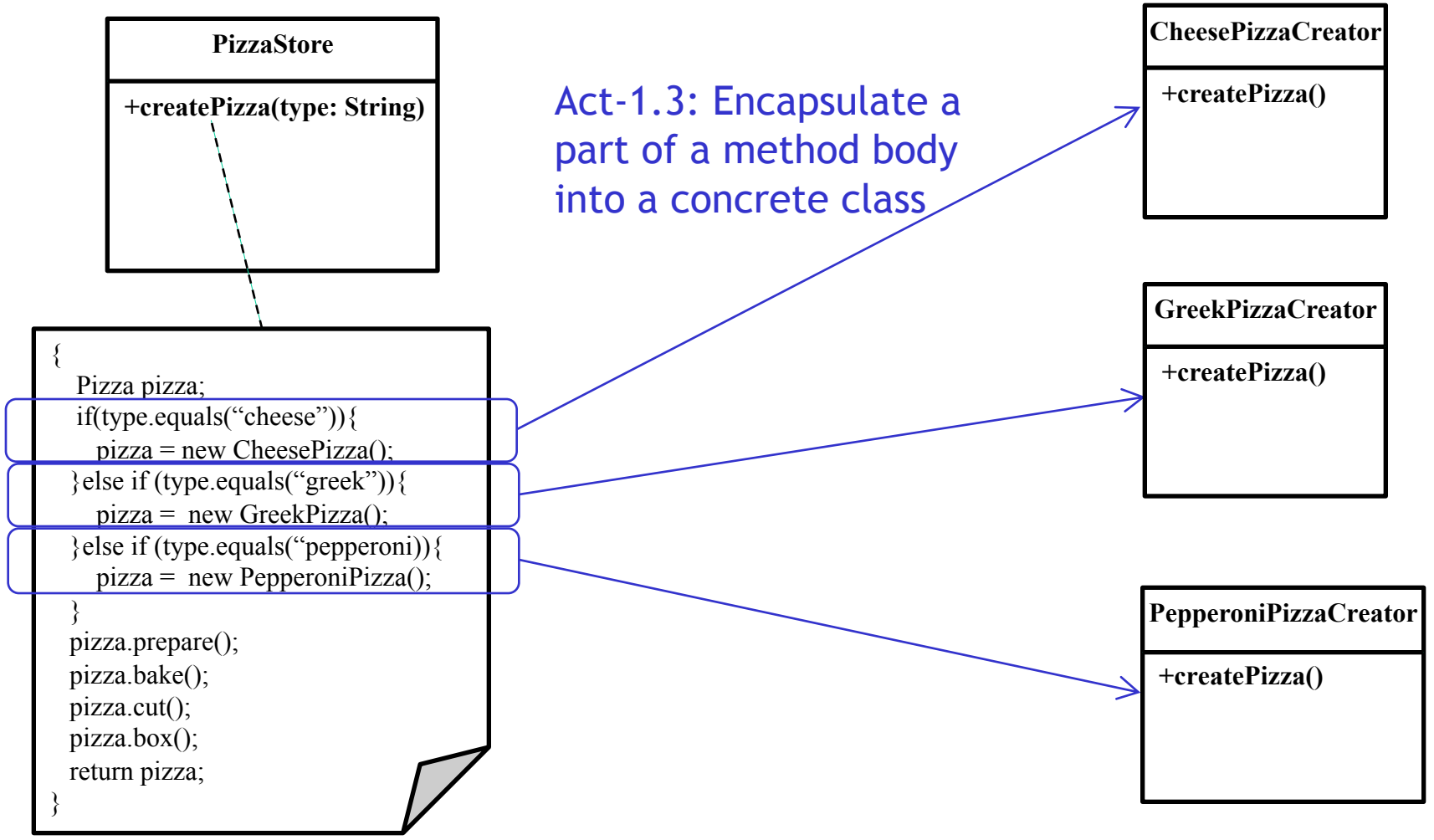
Problems with Initial Design



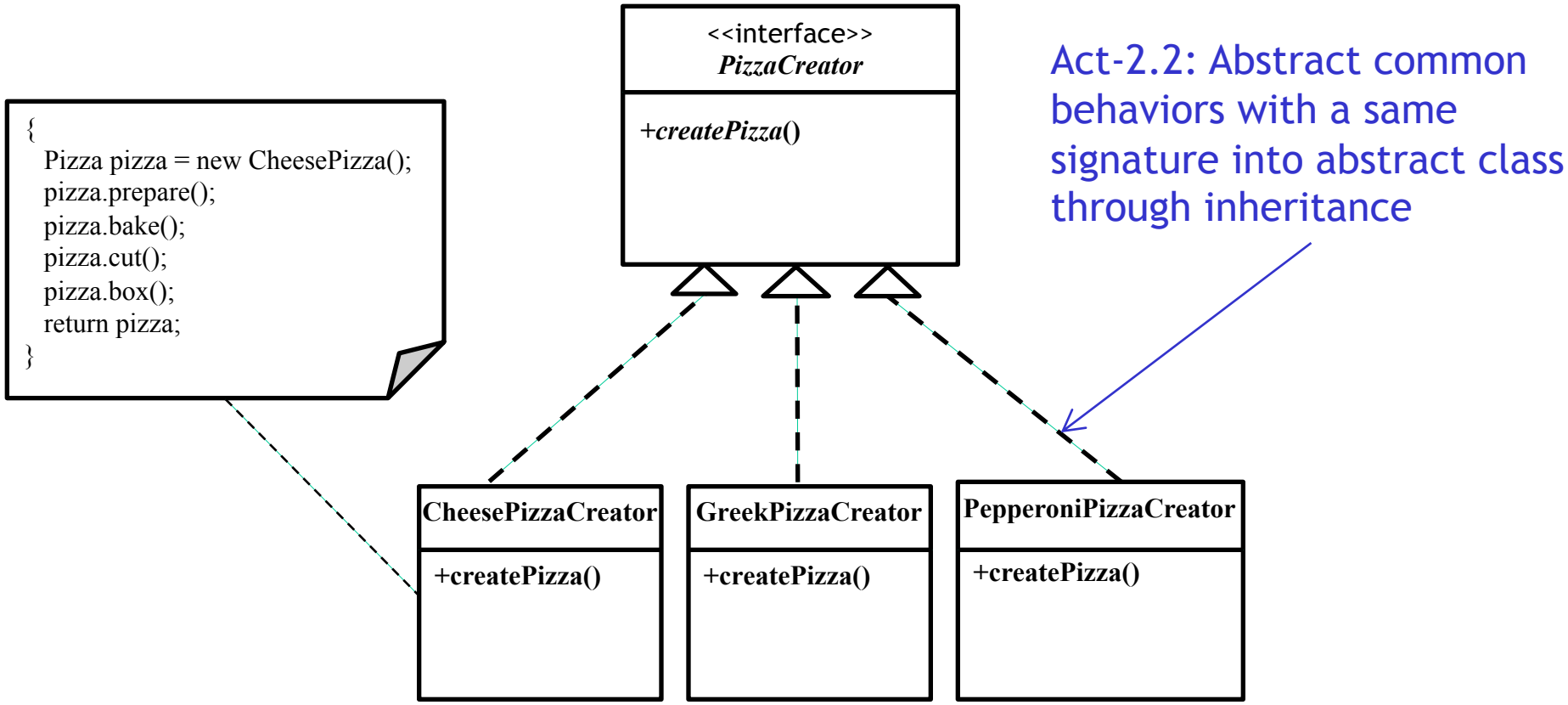
Design Process for Change



Act-1: Encapsulate What Varies

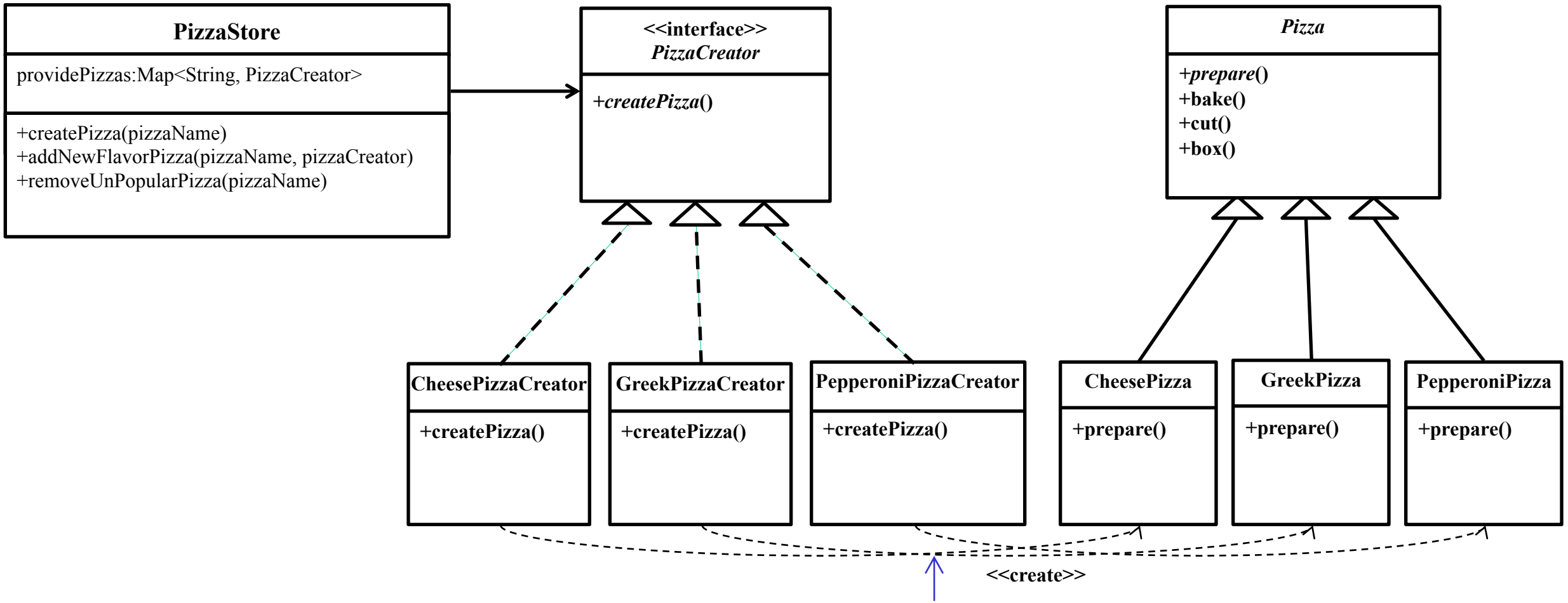


Act-2: Abstract Common Behaviors

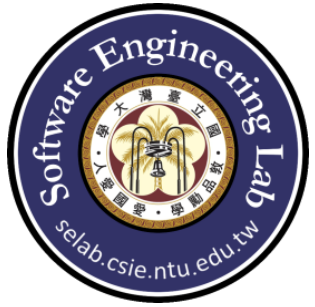


Act-2.2: Abstract common behaviors with a same signature into abstract class through inheritance

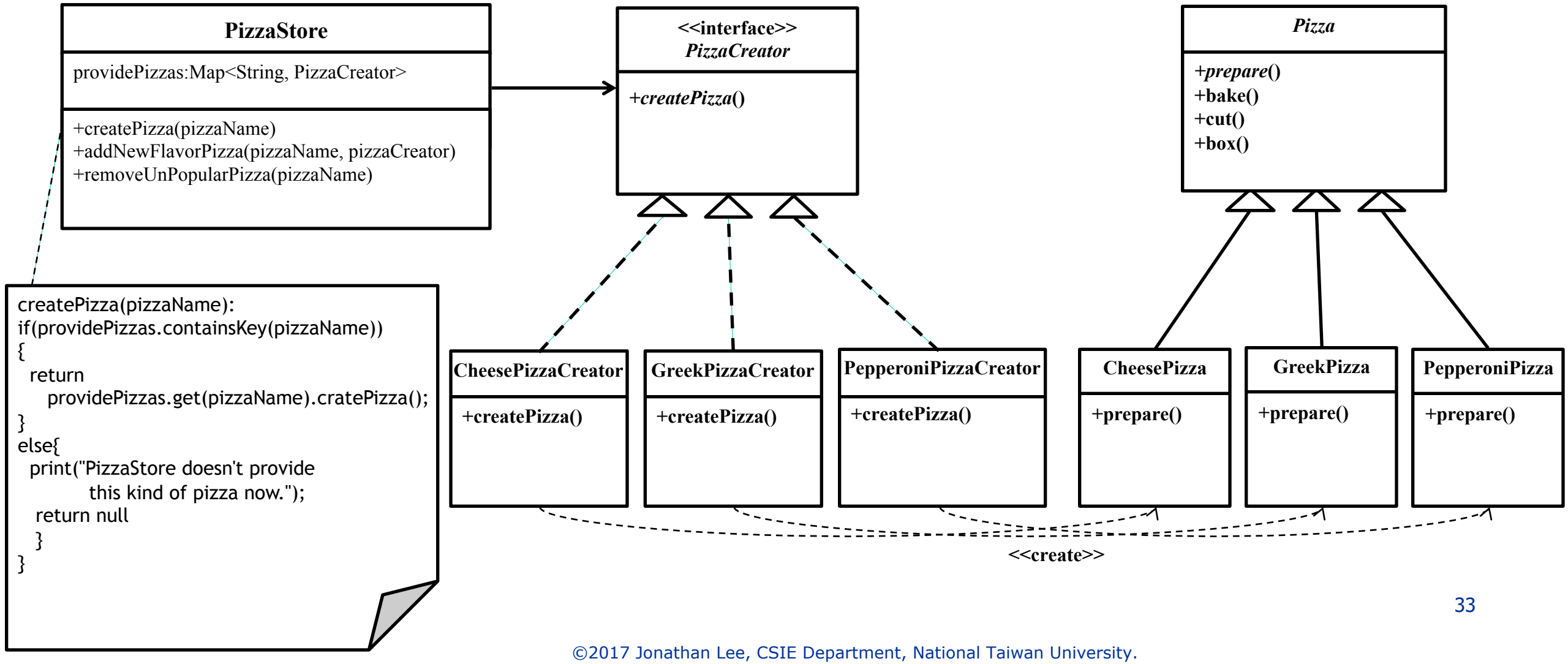
Act-3: Delegate Abstract Behaviors

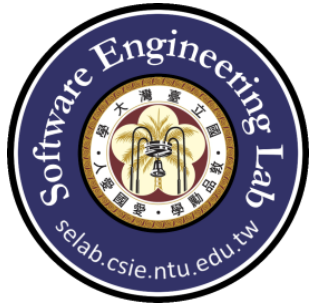


Act-3.4: Delegate behavior to a method of a concrete class



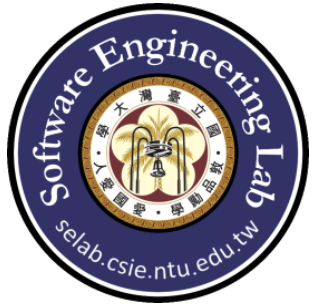
Refactored Design after Design Process





PizzaStore

```
public class PizzaStore {
    private Map<String, PizzaCreator> providePizzas = new HashMap();

    public PizzaStore(){
        addNewFlavorPizza( pizzaName: "Cheese", new CheesePizzaCreator());
        addNewFlavorPizza( pizzaName: "Greek", new GreekPizzaCreator());
        addNewFlavorPizza( pizzaName: "Pepperoni", new PepperoniPizzaCreator());
    }

    public Pizza createPizza(String pizzaName){
        if(providePizzas.containsKey(pizzaName)){
            return providePizzas.get(pizzaName).cratePizza();
        }
        else{
            System.out.println("PizzaStore doesn't provide this kind of pizza now.");
            return null;
        }
    }


    public void addNewFlavorPizza(String pizzaName, PizzaCreator pizzaCreator){
        providePizzas.put(pizzaName, pizzaCreator);
    }

    public void removeUnPopularPizza(String pizzaName) { providePizzas.remove(pizzaName); }

}
```



Source code

PizzaCreator

```
public interface PizzaCreator {

    public Pizza cratePizza();
}
```

CheesePizzaCreator

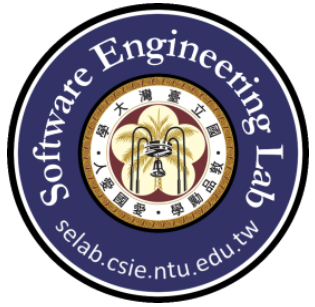
```
public class CheesePizzaCreator implements PizzaCreator {
    @Override
    public Pizza cratePizza() {
        Pizza pizza = new CheesePizza();
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

GreekPizzaCreator

```
public class GreekPizzaCreator implements PizzaCreator {
    @Override
    public Pizza cratePizza() {
        Pizza pizza = new GreekPizza();
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

PepperoniPizzaCreator

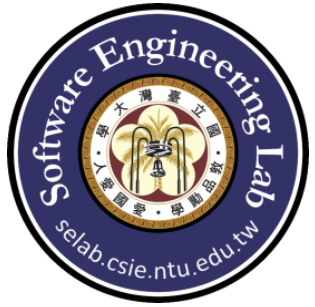
```
public class PepperoniPizzaCreator implements PizzaCreator {
    @Override
    public Pizza cratePizza() {
        Pizza pizza = new PepperoniPizza();
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```



Source code

Pizza

```
public abstract class Pizza {

    public abstract void prepare();

    public void bake() { System.out.println("Bake Pizza"); }

    public void cut() { System.out.println("Cut Pizza"); }

    public void box() { System.out.println("Box Pizza"); }
}
```



Source code

CheesePizza

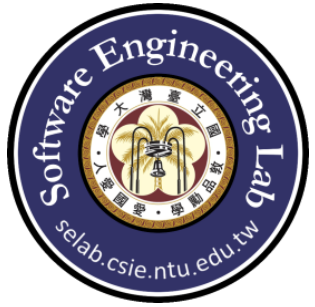
```
public class CheesePizza extends Pizza{
    @Override
    public void prepare() { System.out.println("Prepare Cheese Pizza"); }
}
```

GreekPizza

```
public class GreekPizza extends Pizza{
    @Override
    public void prepare() { System.out.println("Prepare Greek Pizza"); }
}
```

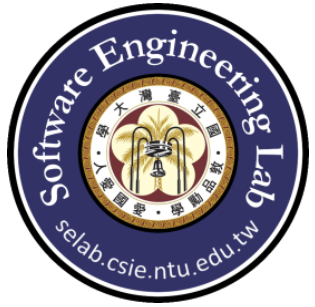
PepperoniPizza

```
public class PepperoniPizza extends Pizza{
    @Override
    public void prepare() { System.out.println("Prepare Pepperoni Pizza"); }
}
```



Recurrent Problem

- As the objects being created changes over time, we need to modify the code of the creator object for the creations over and over again.
  - We need to encapsulate the knowledge of which objects to create and moves this knowledge out of the creator object.



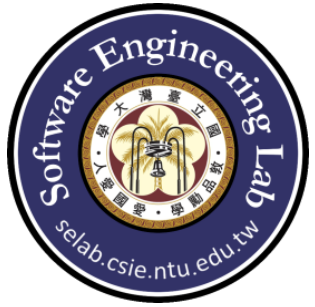
Input/Output

Input:

```
Create [Pizza_type]

RemoveUnpopularPizza [Pizza_type]

AddNewFlavorPizza [Pizza_type]

...
```

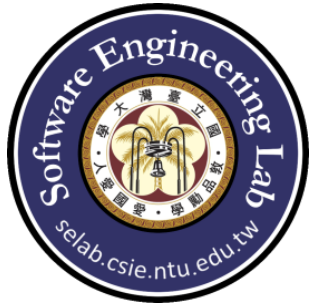
Output:

```
//if [Pizza_type] pizza is not provided

PizzaStore doesn't provide this kind of pizza now.


// if [Pizza_type]  pizza is provided

Prepare [Pizza_type] Pizza

Bake Pizza

Cut Pizza

Box Pizza

...
```



Test cases

- TestCase 1: Create 3 kinds of Pizza
- TestCase 2: Remove 3 kinds of Pizza
- TestCase 3: Add 3 kinds of Pizza



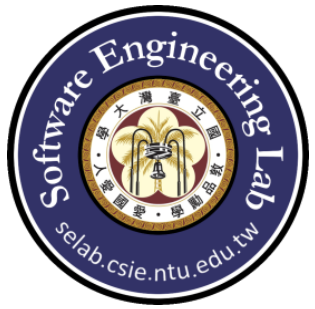
Test case1

Sample1.in

```
1  Create Cheese
2  Create Greek
3  Create Pepperoni
```

Sample1.out

```
1  Prepare Cheese Pizza
2  Bake Pizza
3  Cut Pizza
4  Box Pizza
5  Prepare Greek Pizza
6  Bake Pizza
7  Cut Pizza
8  Box Pizza
9  Prepare Pepperoni Pizza
10 Bake Pizza
11 Cut Pizza
12 Box Pizza
```



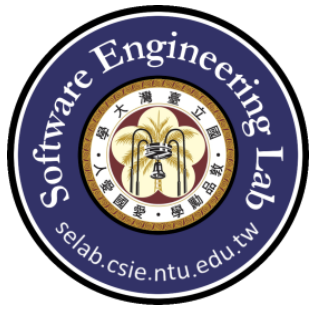
Test case2

Sample2.in

```
1  Create Cheese
2  RemoveUnpopularPizza Cheese
3  Create Cheese
4  Create Pepperoni
5  RemoveUnpopularPizza Pepperoni
6  Create Pepperoni
7  Create Greek
8  RemoveUnpopularPizza Pepperoni
9  Create Greek
```

Sample2.out

```
1  Prepare Cheese Pizza
2  Bake Pizza
3  Cut Pizza
4  Box Pizza
5  PizzaStore doesn't provide this kind of pizza now.
6  Prepare Pepperoni Pizza
7  Bake Pizza
8  Cut Pizza
9  Box Pizza
10 PizzaStore doesn't provide this kind of pizza now.
11 Prepare Greek Pizza
12 Bake Pizza
13 Cut Pizza
14 Box Pizza
15 Prepare Greek Pizza
16 Bake Pizza
17 Cut Pizza
18 Box Pizza
```



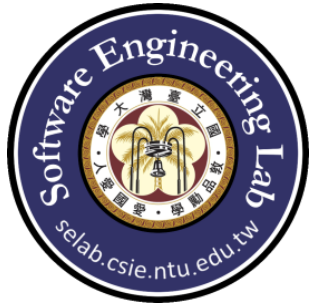
Test case3

Sample3.in

```
1  Create Cheese
2  RemoveUnpopularPizza Cheese
3  Create Cheese
4  AddNewFlavorPizza Cheese
5  Create Cheese
6  Create Pepperoni
7  RemoveUnpopularPizza Pepperoni
8  Create Pepperoni
9  AddNewFlavorPizza Pepperoni
10 Create Pepperoni
11 Create Greek
12 RemoveUnpopularPizza Pepperoni
13 Create Greek
14 AddNewFlavorPizza Greek
15 Create Greek
```

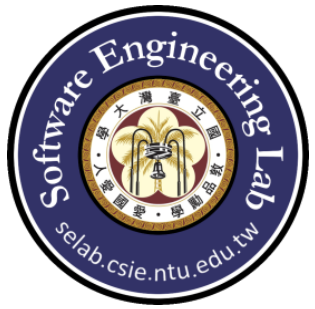
Sample3.out

```
1  Prepare Cheese Pizza
2  Bake Pizza
3  Cut Pizza
4  Box Pizza
5  PizzaStore doesn't provide this kind of pizza now.
6  Prepare Cheese Pizza
7  Bake Pizza
8  Cut Pizza
9  Box Pizza
10 Prepare Pepperoni Pizza
11 Bake Pizza
12 Cut Pizza
13 Box Pizza
14 PizzaStore doesn't provide this kind of pizza now.
15 Prepare Pepperoni Pizza
16 Bake Pizza
17 Cut Pizza
18 Box Pizza
19 Prepare Greek Pizza
20 Bake Pizza
21 Cut Pizza
22 Box Pizza
23 Prepare Greek Pizza
24 Bake Pizza
25 Cut Pizza
26 Box Pizza
27 Prepare Greek Pizza
28 Bake Pizza
29 Cut Pizza
30 Box Pizza
```

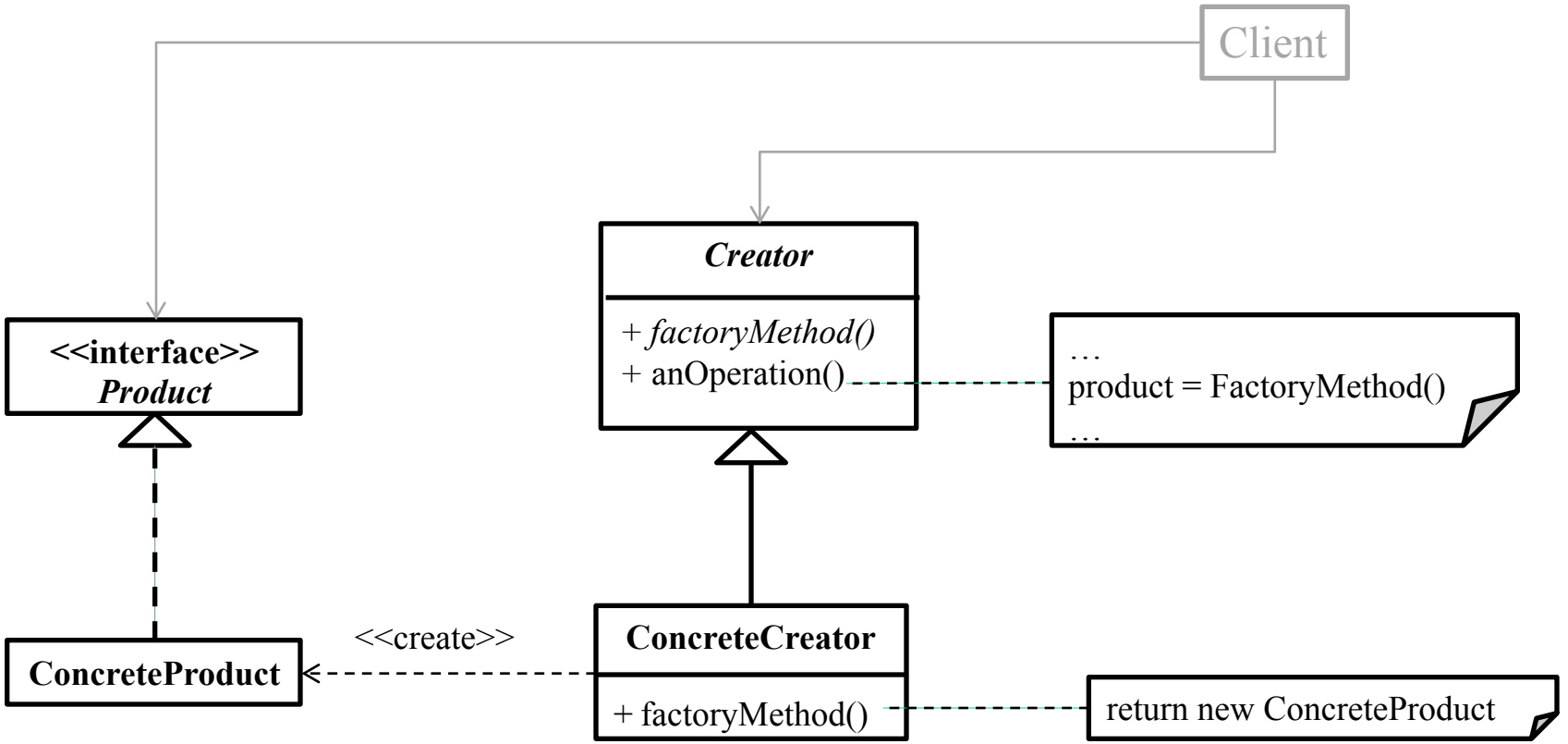


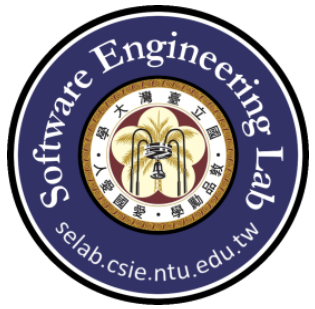
Factory Method Pattern

- Intent
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

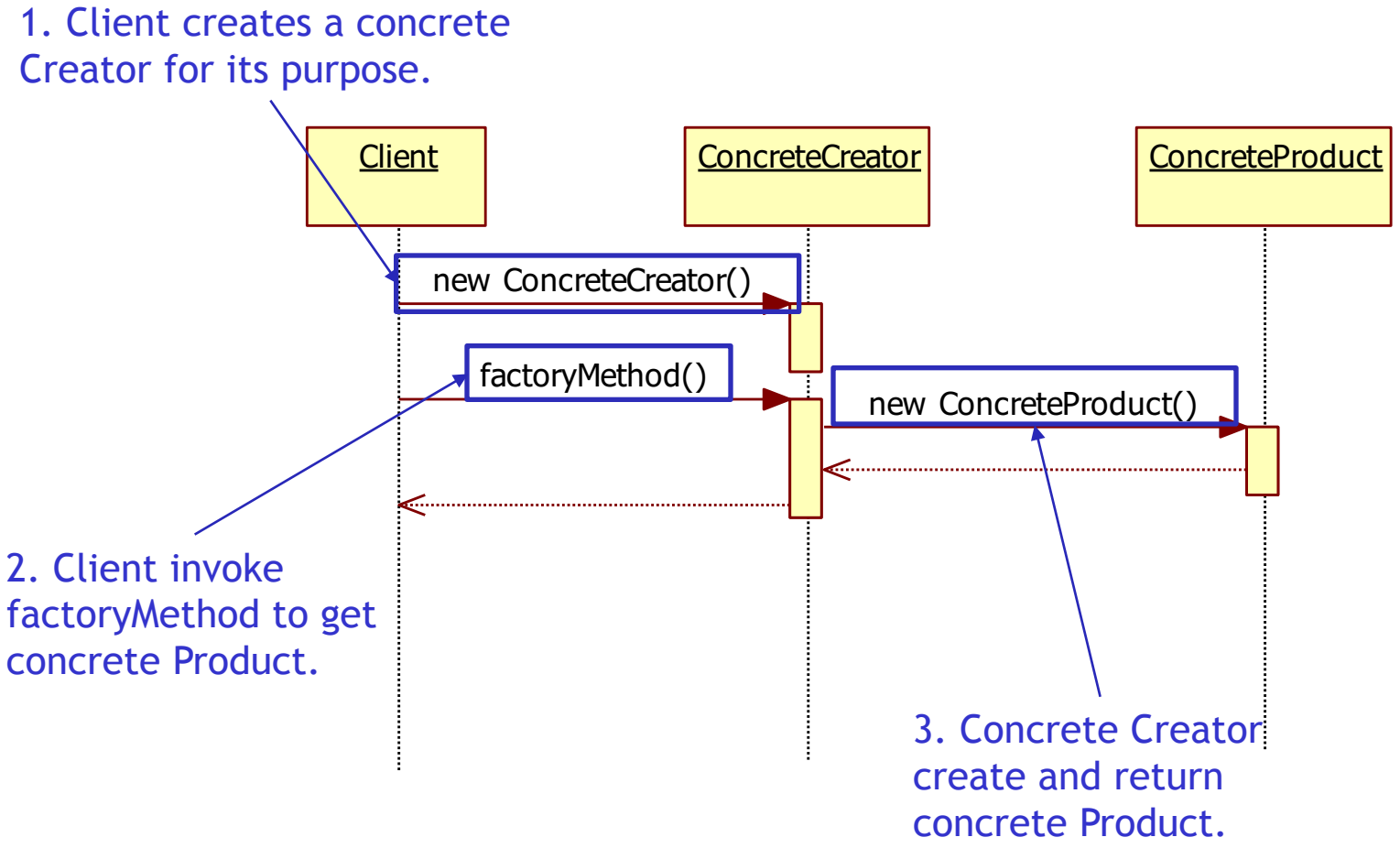


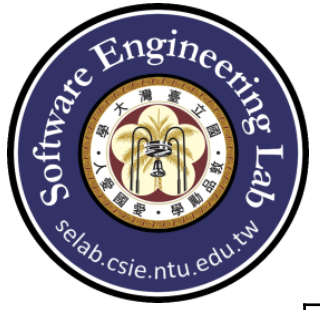
Factory Method Pattern Structure$_1$





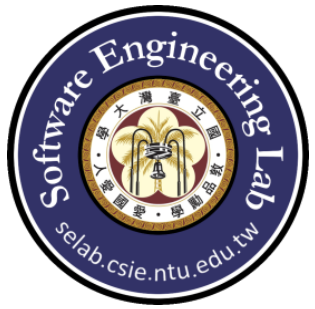
Factory Method Pattern Structure$_2$



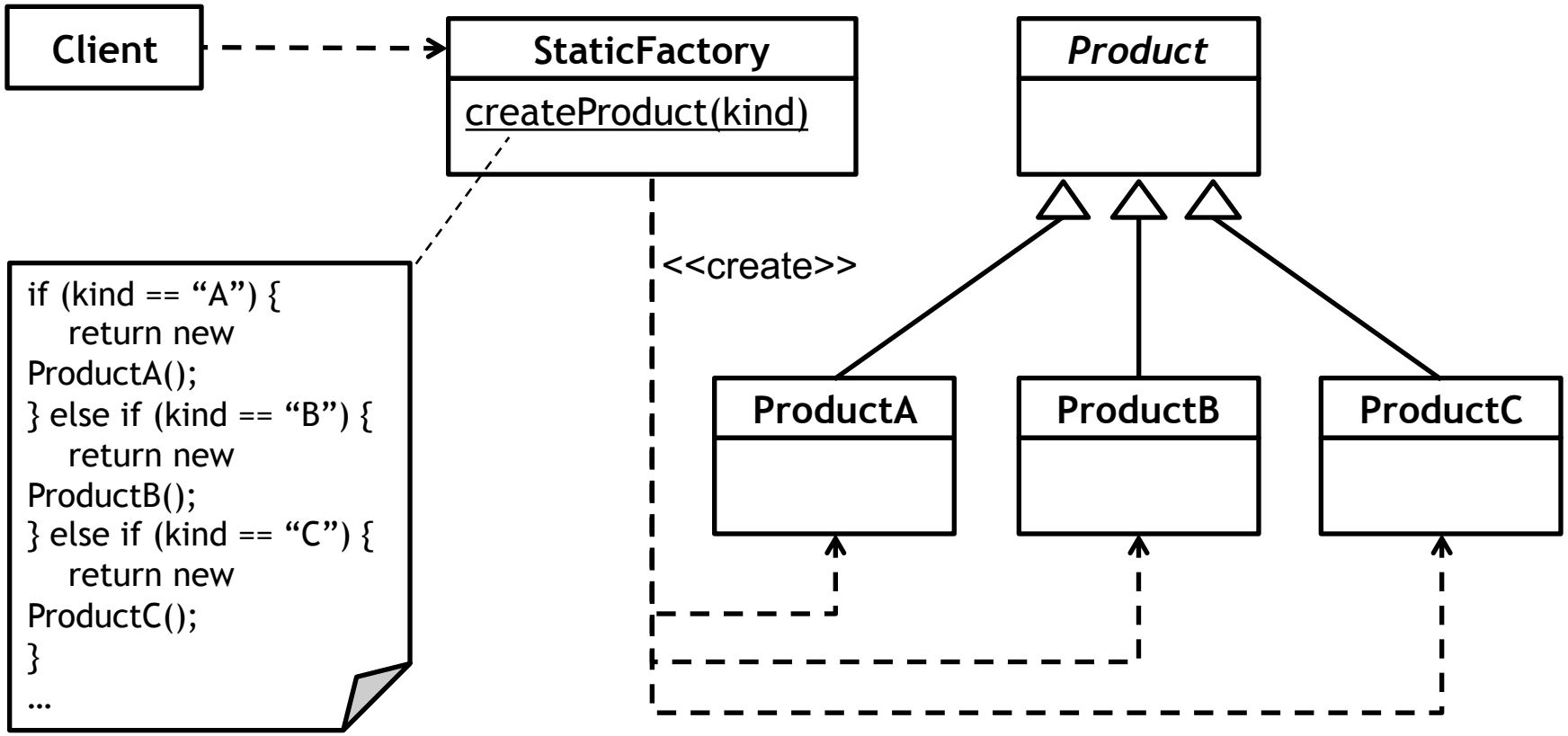


Factory Method Pattern Structure$_3$

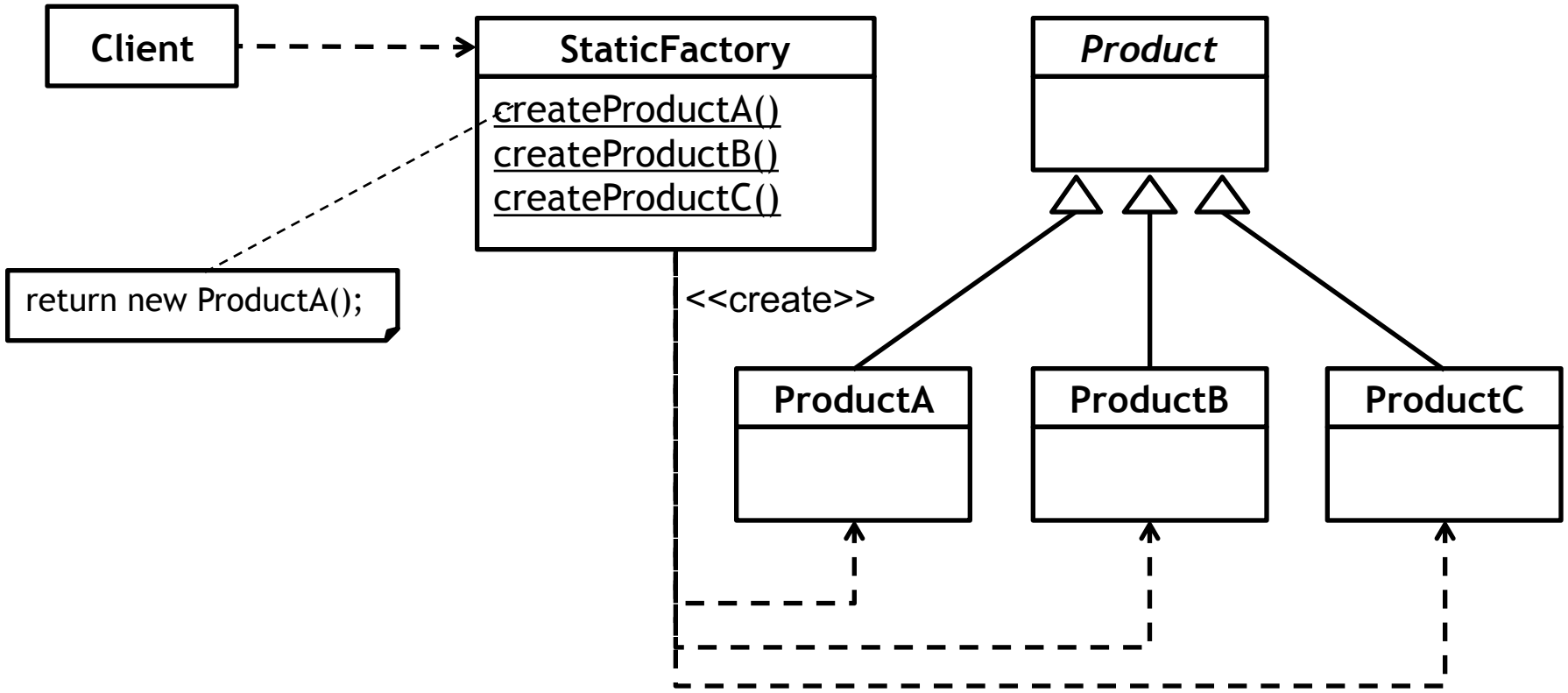
	Instantiation	Use	Termination
Client	Other class except classes in the factory method	Other class except classes in the factory method	Other class except classes in the factory method
Product	X	Client class use Concrete Product through this interface	X
Concrete Product	**Concrete Creator**	Client class	Other class or the client class
Creator	X	Client class use this interface to get product that produced by Concrete Creator	X
Concrete Creator	Other class or the client class	Client class	Other class or the client class

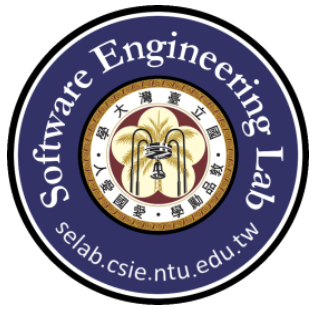


Static Factory (I)

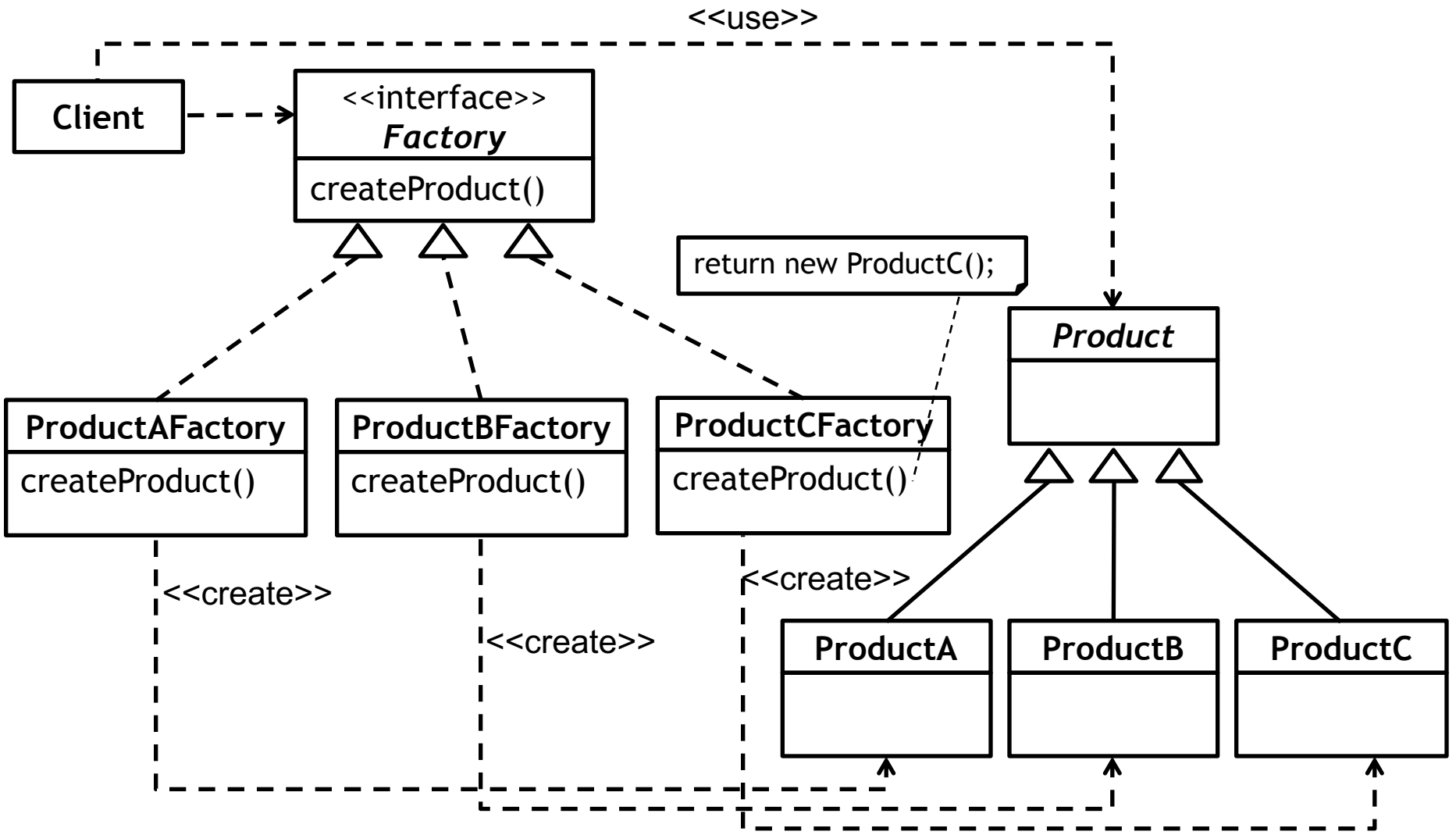


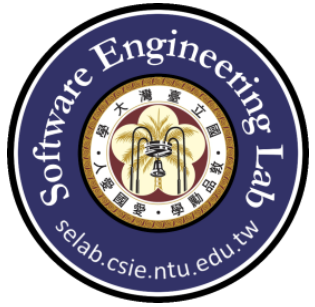
Static Factory (II)





Non-Static Factory





Static Factory vs. Non-Static Factory

	Static Factory	Non-Static Factory
Instantiation	You don't need to instantiate a factory object for creation.	You have to instantiate a specific factory object to create specific products.
Overriding	You can't override the factory method because it is impossible to override a static method.	You can override the factory method through subclassing.
Relationship	The factory class must know all classes that it is in charge of creating.	The factory abstraction just need to know product abstraction instead of every product implementation.
Add New Products	Open factory class and add new if-else statements or new factory methods. (It violates the Open-Closed Principle)	Add new concrete classes and implement the factory interface or inherit the factory abstract class.