Iterator Pattern

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

Design Aspect of Iterator

How an aggregate’s elements are accessed, traversed

Outline

- ❑ Requirements Statement
- ❑ Initial Design and Its Problems
- ❑ Design Process
- ❑ Refactored Design after Design Process
- ❑ Recurrent Problems
- ❑ Intent
- ❑ Iterator Pattern Structure
- ❑ Two kinds of Iterator
- ❑ Another Example
- ❑ Homework

Print Out Items in Different Data Structures (Iterator)

Prof. Jonathan Lee (李允中)

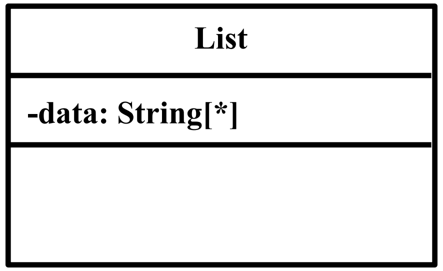
Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statements

- A List data structure is implemented with a String array which can contain a series of String objects.
- We can access List by calling the get() method with an index, and know how many Strings inside the List with a public attribute: length.
- Furthermore, another data structure called SkipList which consists of a series of SkipNodes.
- Each SkipNode can be accessed by invoking the getNode() method in SkipList with an index. And we have the idea about the size of SkipList with its size() method.
- Now we have to traverse both List and SkipList to print out those object items in the two different data structures.

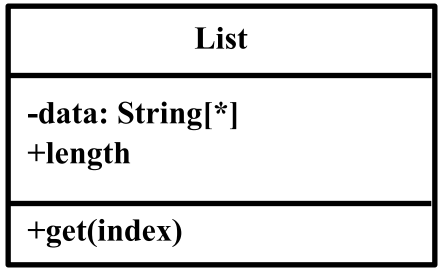
Requirements Statement$_1$

- A List data structure is implemented with a String array which can contain a series of String objects.



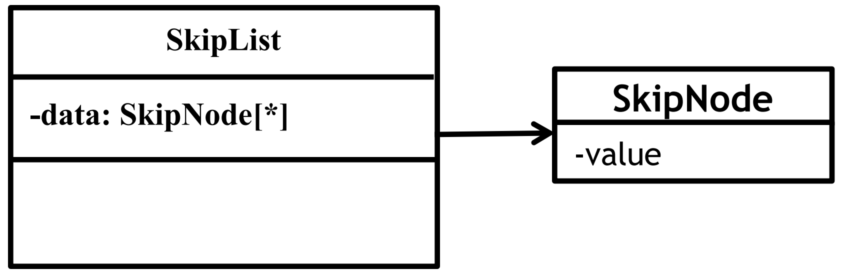
Requirements Statement$_2$

- We can access List by calling the get() method with an index, and know how many Strings inside the List with a public attribute: length.



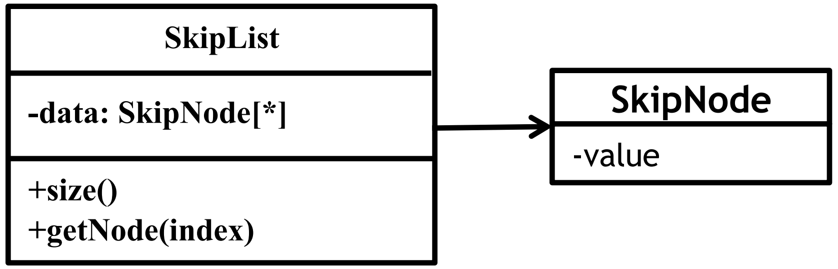
Requirements Statement$_3$

- Furthermore, another data structure called SkipList which consists of a series of SkipNodes.



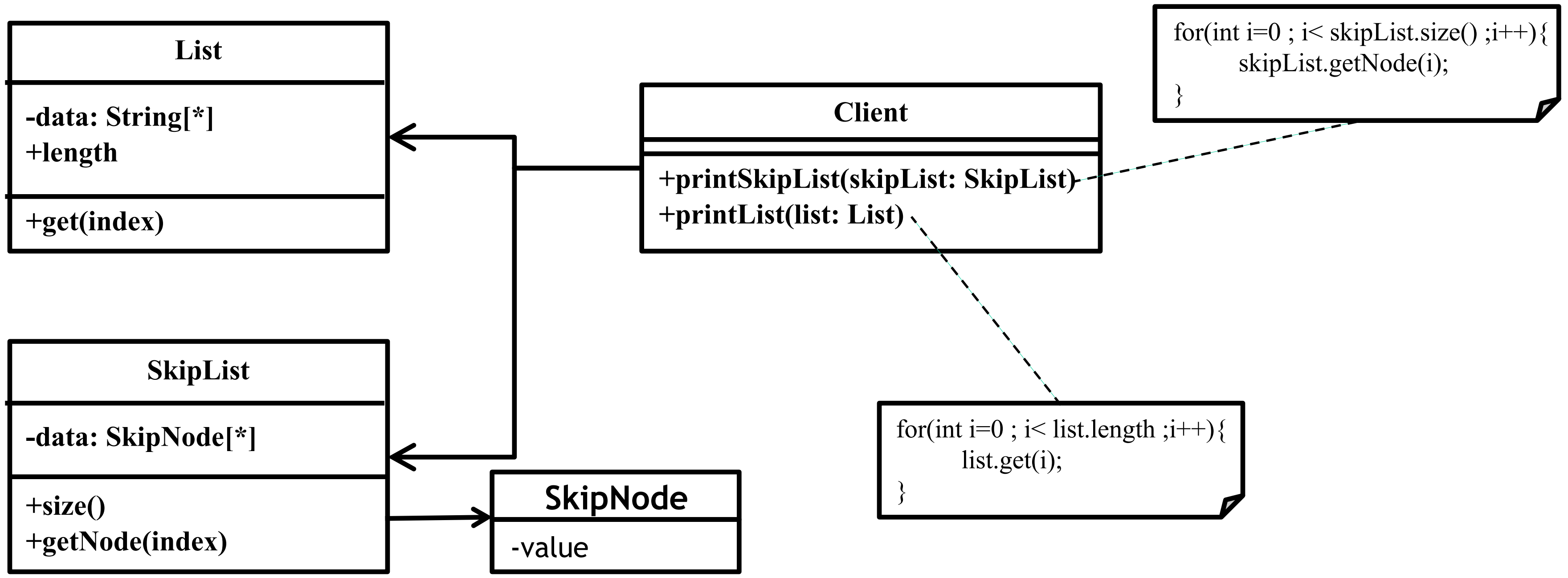
Requirements Statement$_4$

- Each SkipNode can be accessed by invoking the getNode() method in SkipList with an index. And we have the idea about the size of SkipList with its size() method.



Requirements Statement$_5$

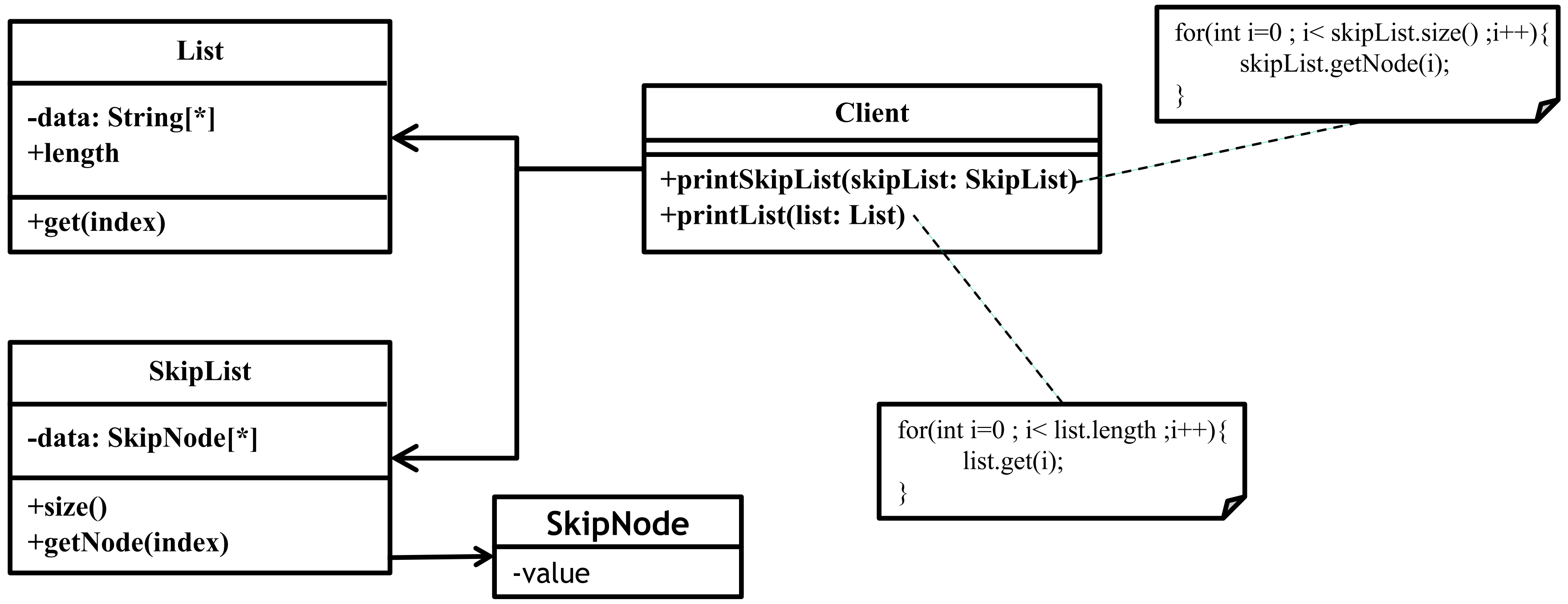
- Now we have to traverse both List and SkipList to print out those object items in the two different data structures.



```
for(int i=0 ; i< skipList.size() ;i++){
        skipList.getNode(i);
}
```

```
for(int i=0 ; i< list.length ;i++){
        list.get(i);
}
```

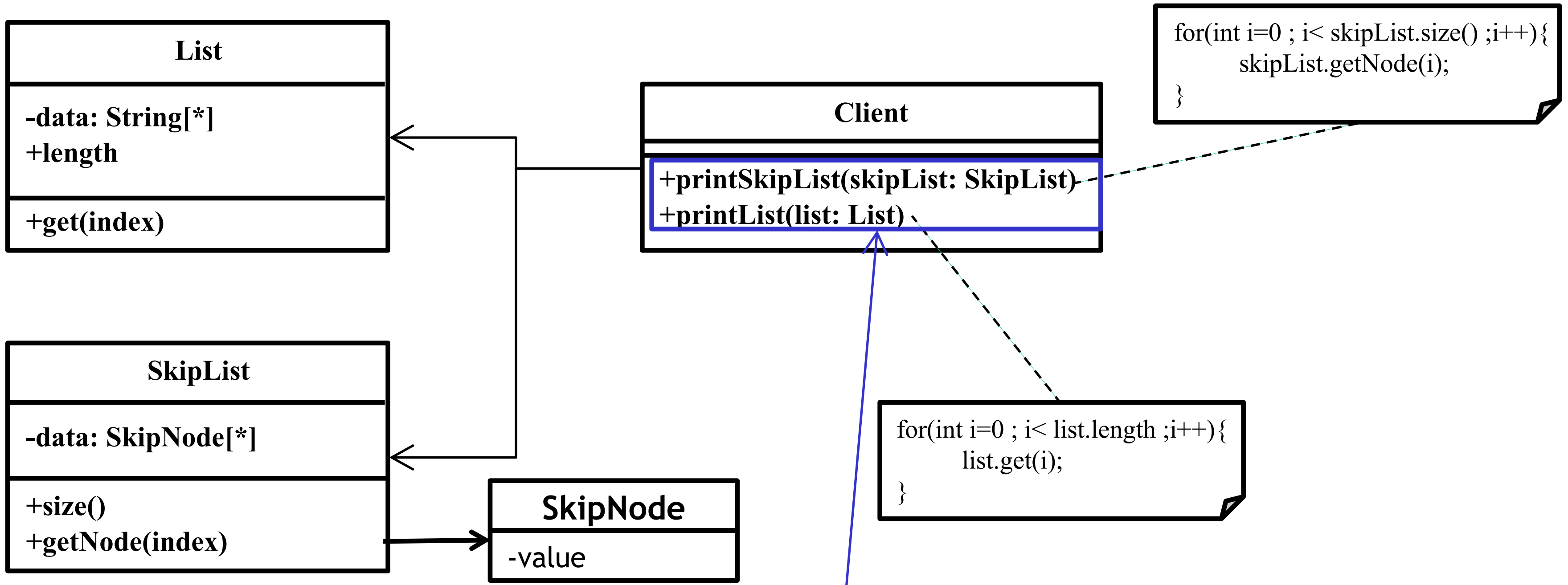
Initial Design



```
for(int i=0 ; i< skipList.size() ;i++){
        skipList.getNode(i);
}
```

```
for(int i=0 ; i< list.length ;i++){
        list.get(i);
}
```

Problems with Initial Design

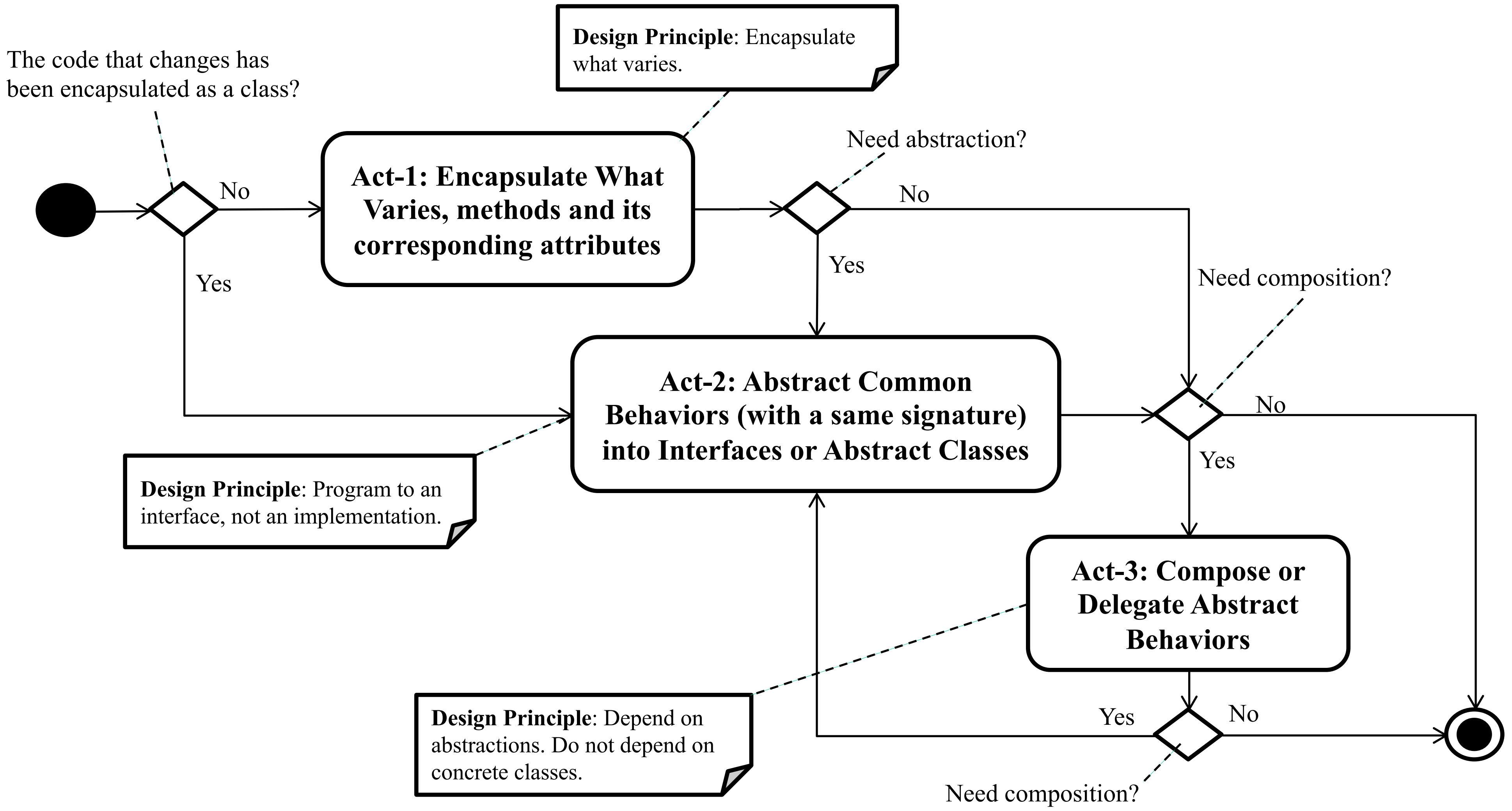


```
for(int i=0 ; i< skipList.size() ;i++){
        skipList.getNode(i);
}
```

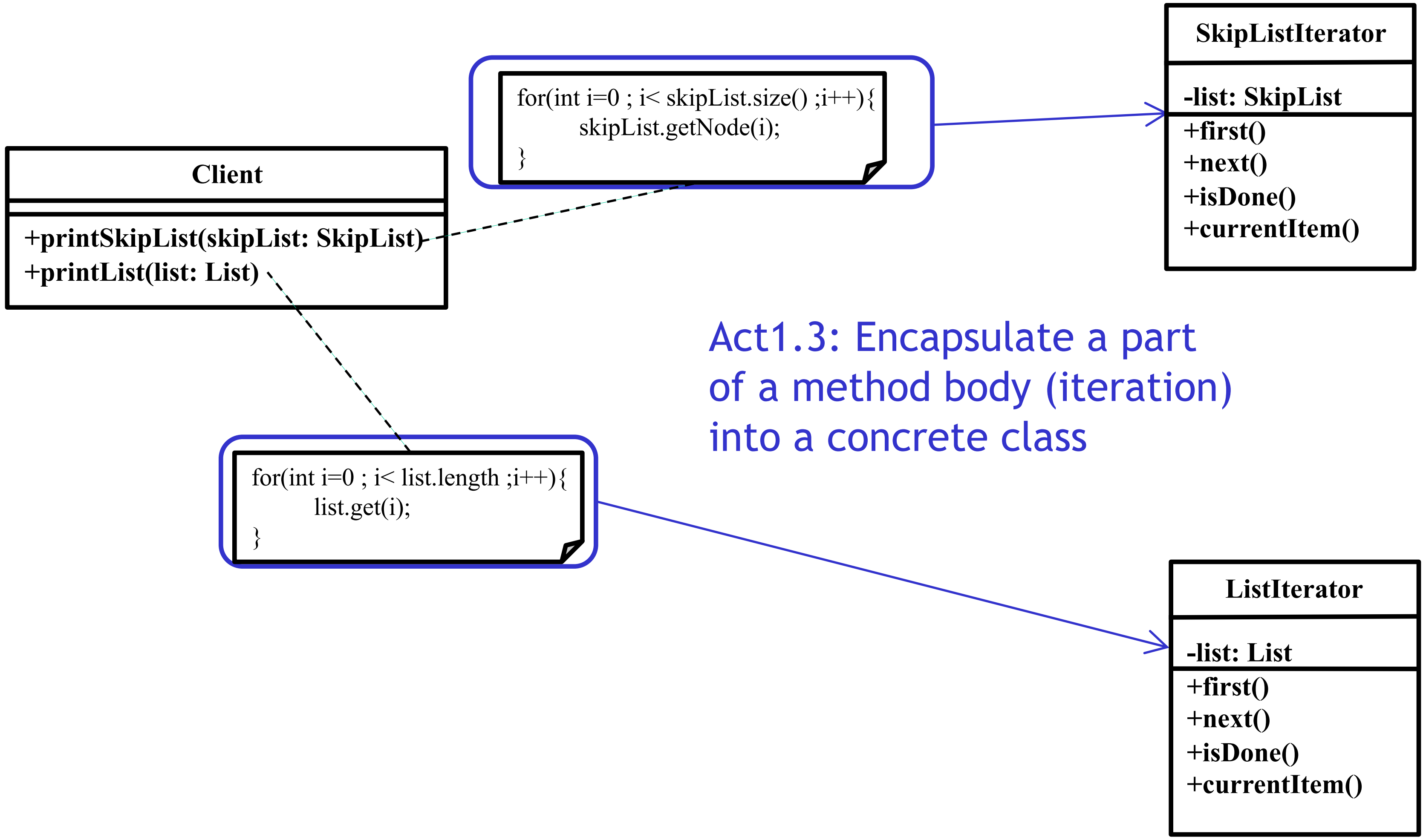
```
for(int i=0 ; i< list.length ;i++){
        list.get(i);
}
```

Problem: We will need to add new print methods if we want to print out all the items in new data structures. And there are some common behaviors in traversing the data structures.

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-1: Encapsulate What Varies

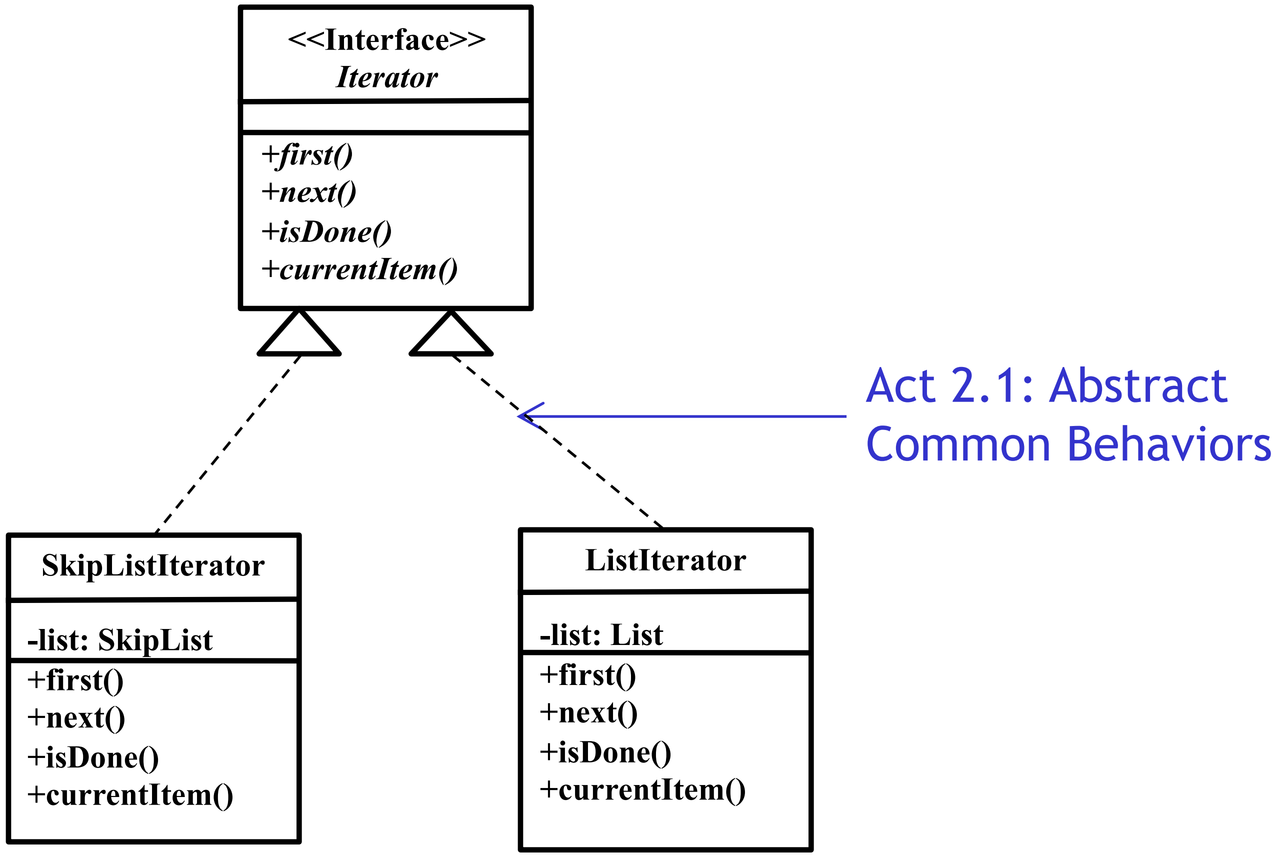


```
for(int i=0 ; i< skipList.size() ;i++){
        skipList.getNode(i);
}
```

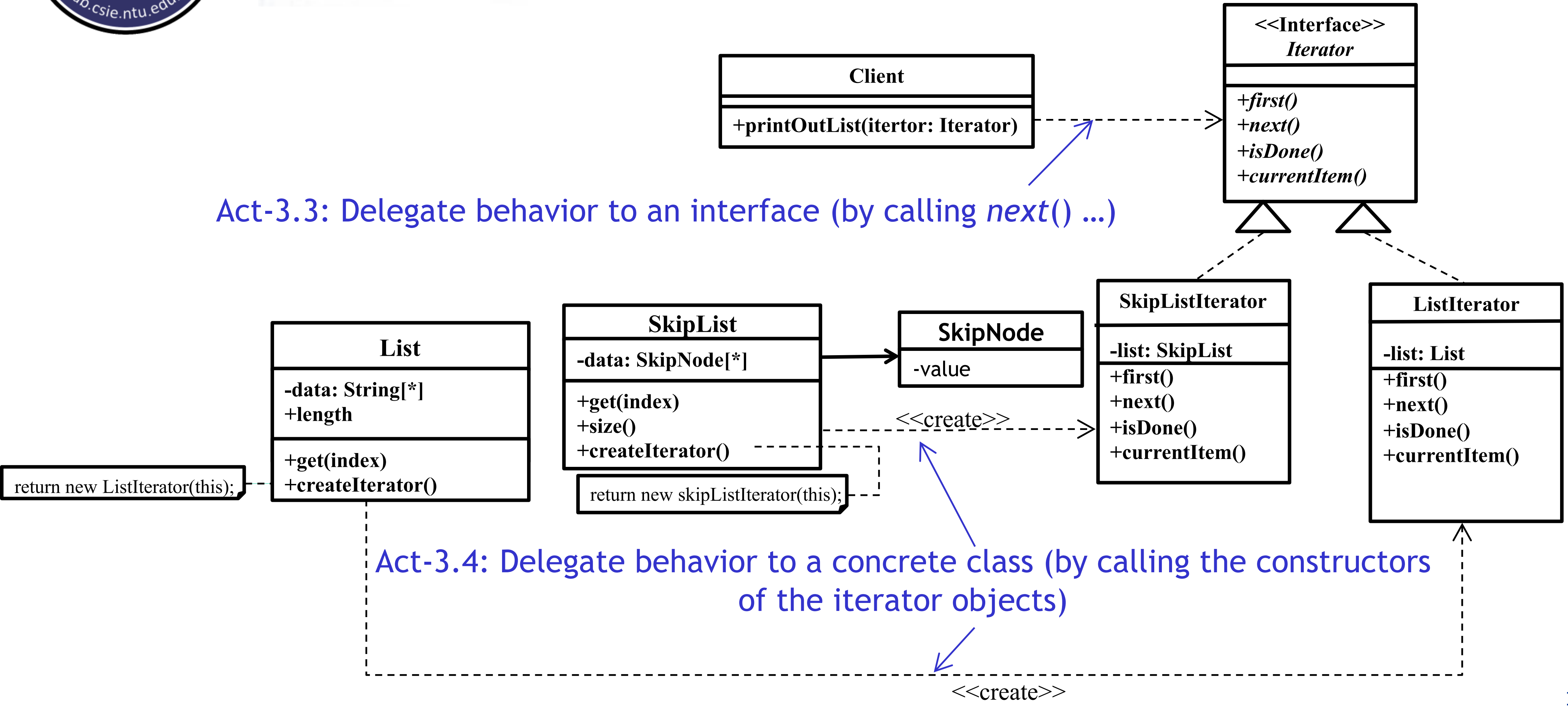
```
for(int i=0 ; i< list.length ;i++){
        list.get(i);
}
```

Act1.3: Encapsulate a part of a method body (iteration) into a concrete class

Act-2: Abstract Common Behaviors



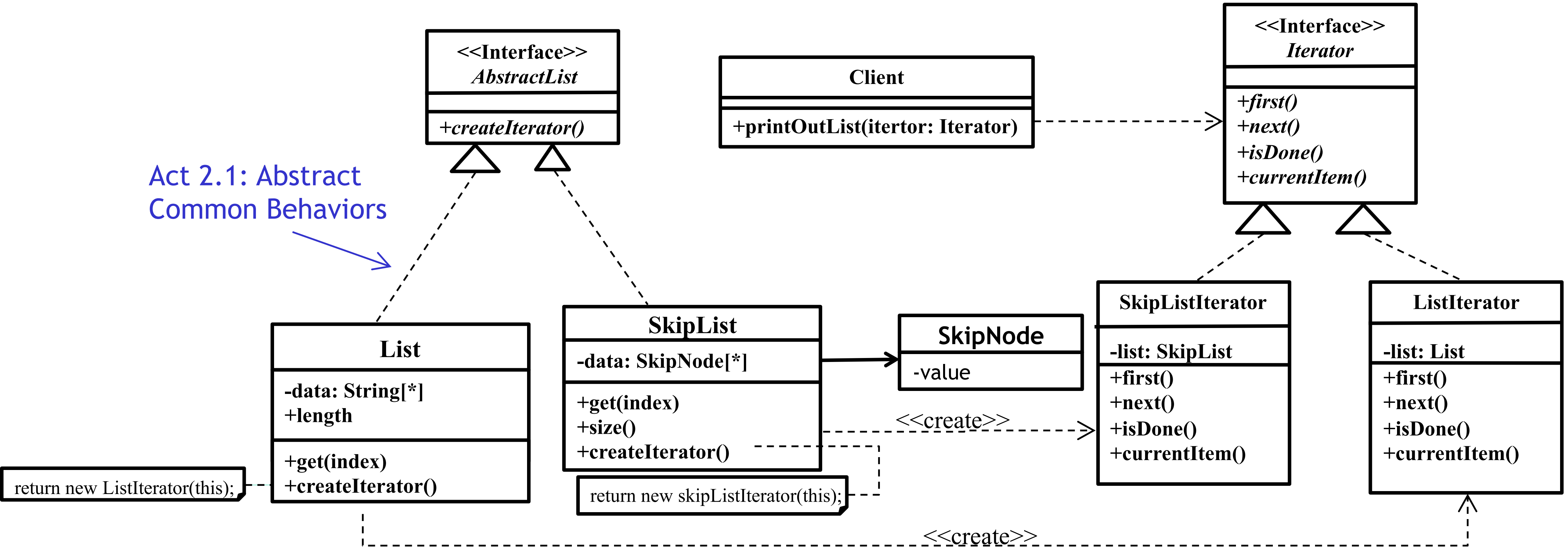
Act-3: Compose Abstract Interfaces/Abstract Classes or Delegate Behaviors



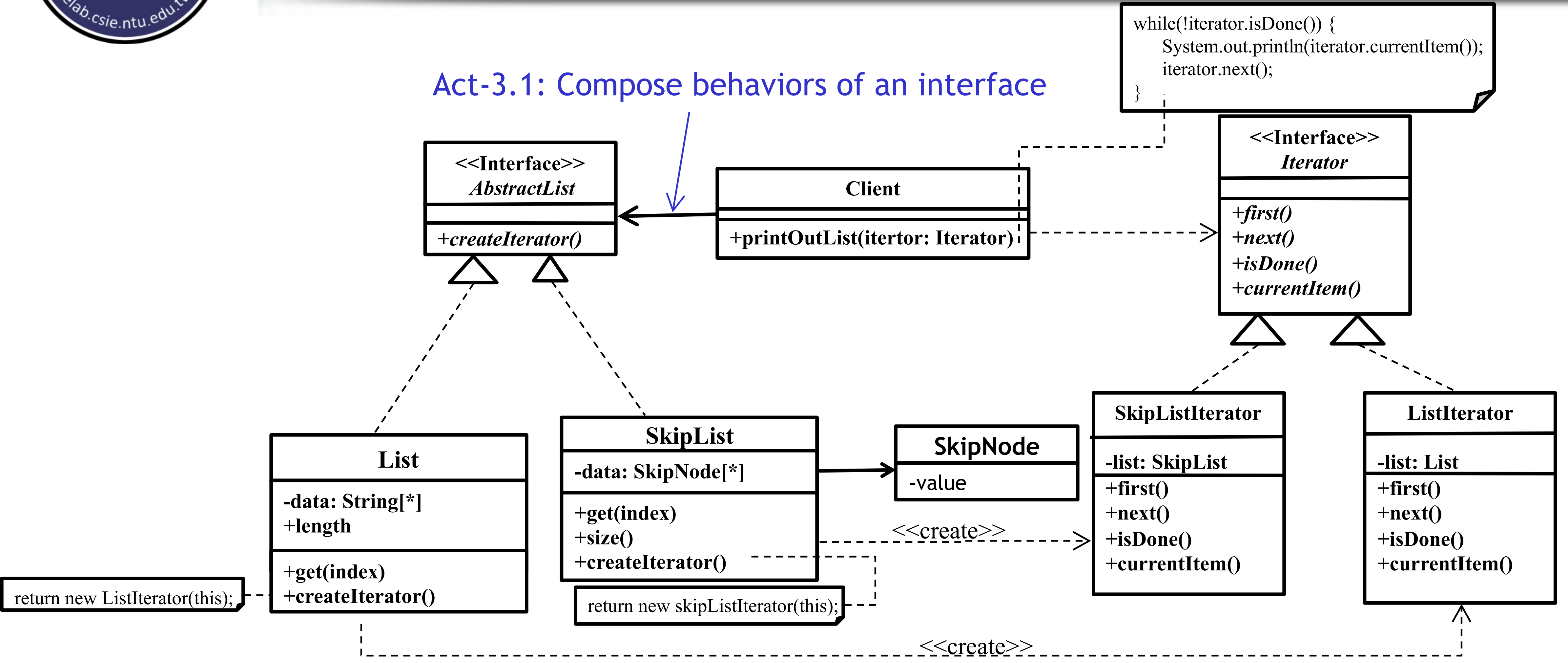
Act-3.3: Delegate behavior to an interface (by calling *next*() …)

Act-3.4: Delegate behavior to a concrete class (by calling the constructors of the iterator objects)

Act-2: Abstract Common Behaviors

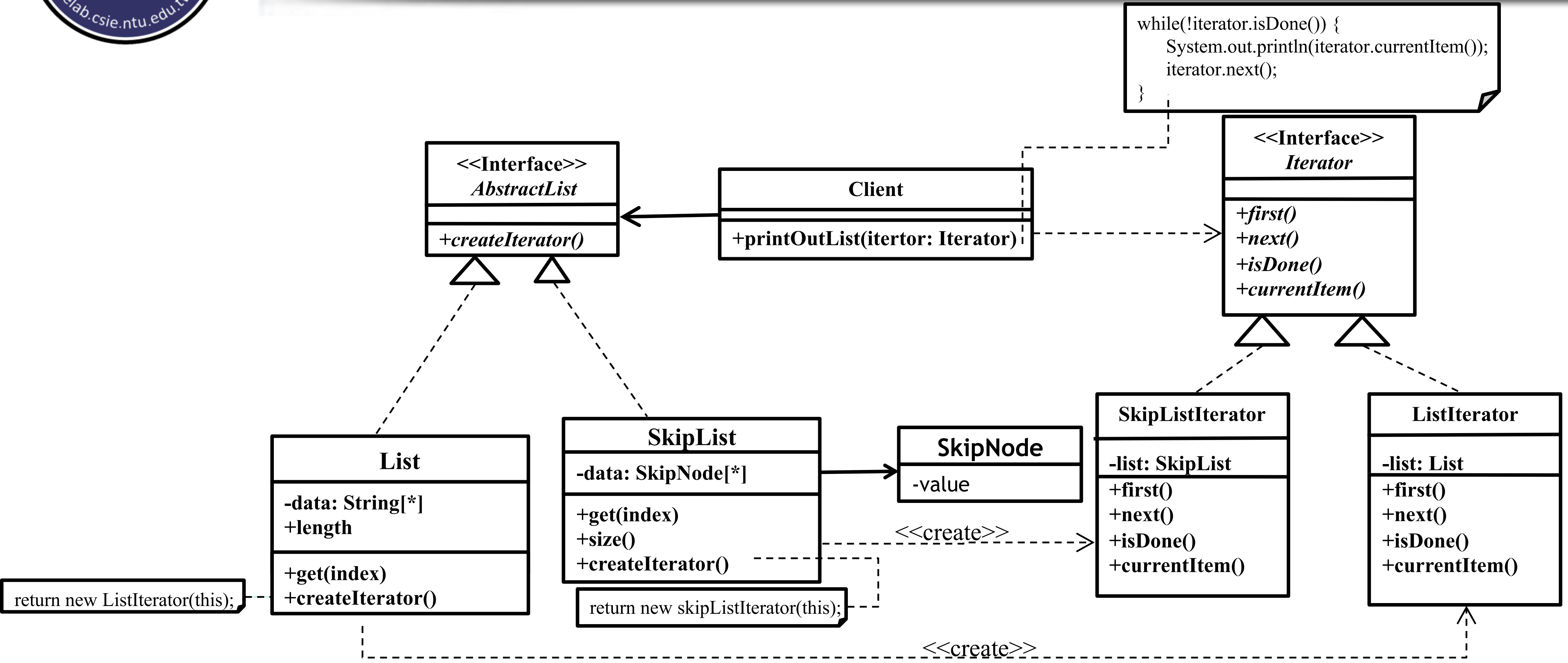


Act-3: Compose Abstract Interfaces/Abstract Classes or Delegate Behaviors



```
while(!iterator.isDone()) {
    System.out.println(iterator.currentItem());
    iterator.next();
}
```

Refactored Design after Design Process

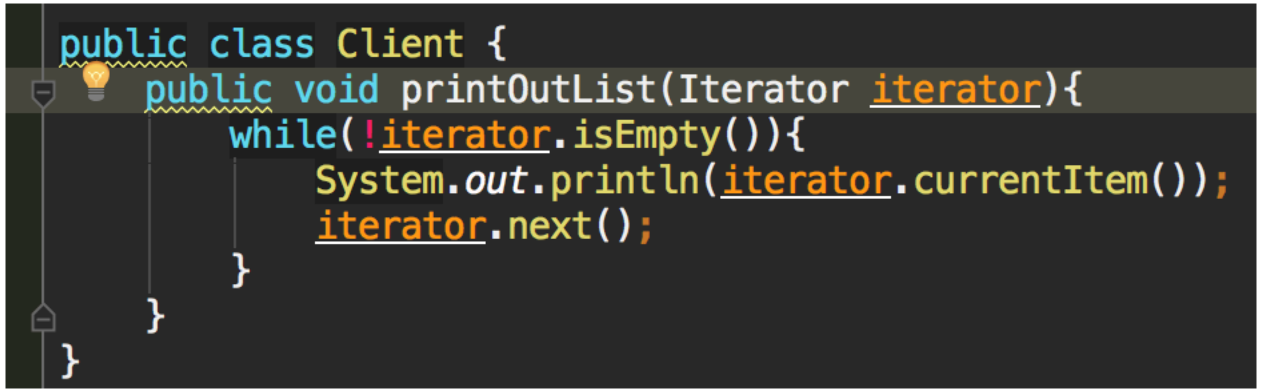


```
while(!iterator.isDone()) {
    System.out.println(iterator.currentItem());
    iterator.next();
}
```

```
return new ListIterator(this);
```

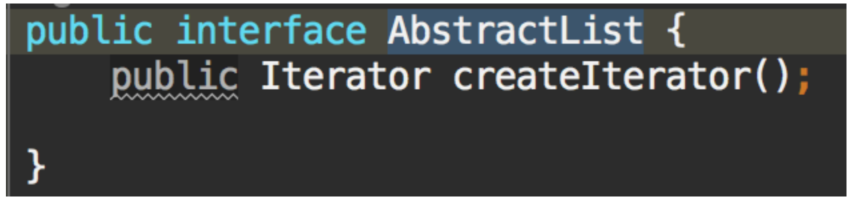
```
return new skipListIterator(this);
```

Client



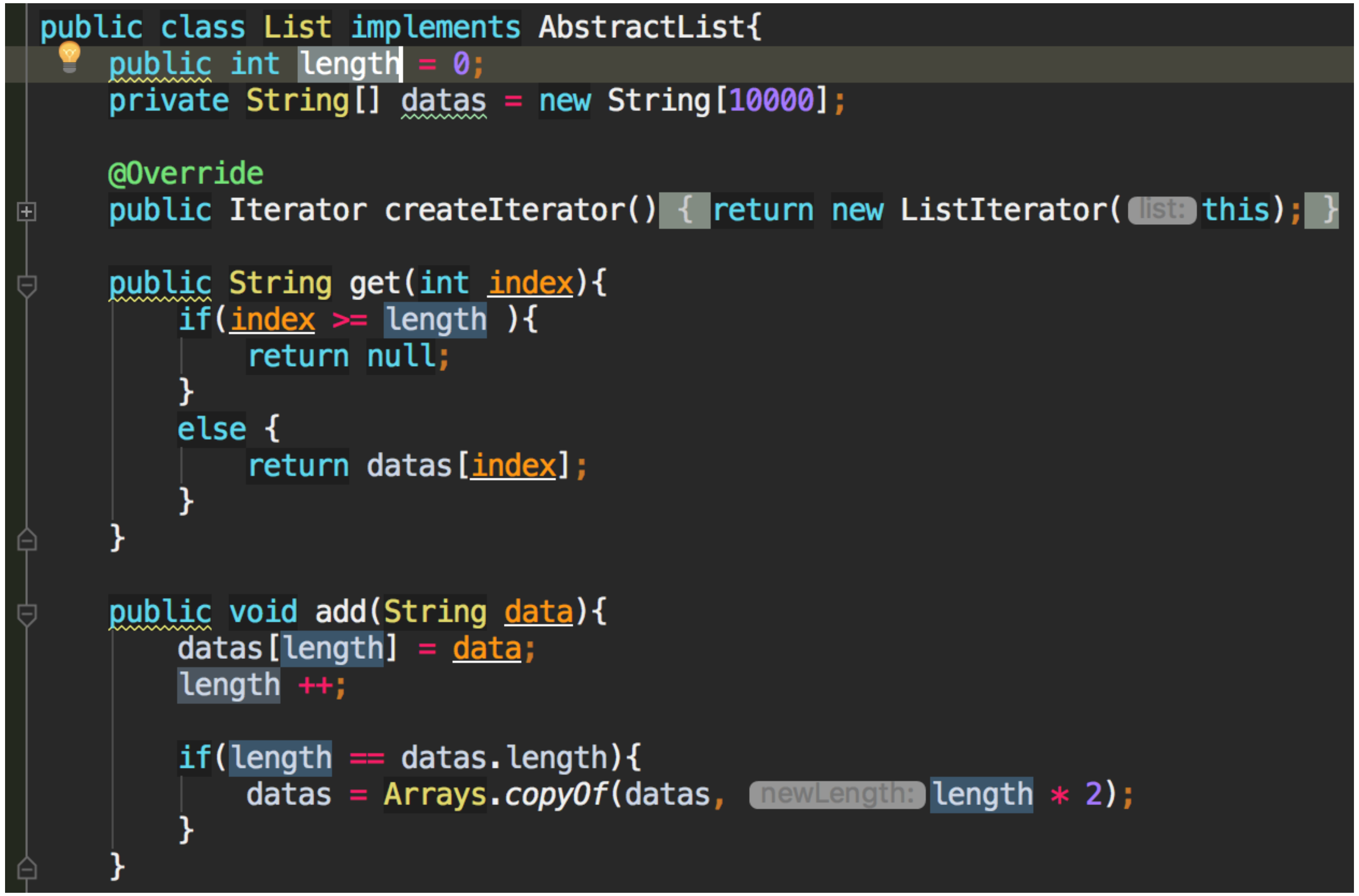
```java
public class Client {
    public void printOutList(Iterator iterator){
        while(!iterator.isEmpty()){
            System.out.println(iterator.currentItem());
            iterator.next();
        }
    }
}
```

AbstractList



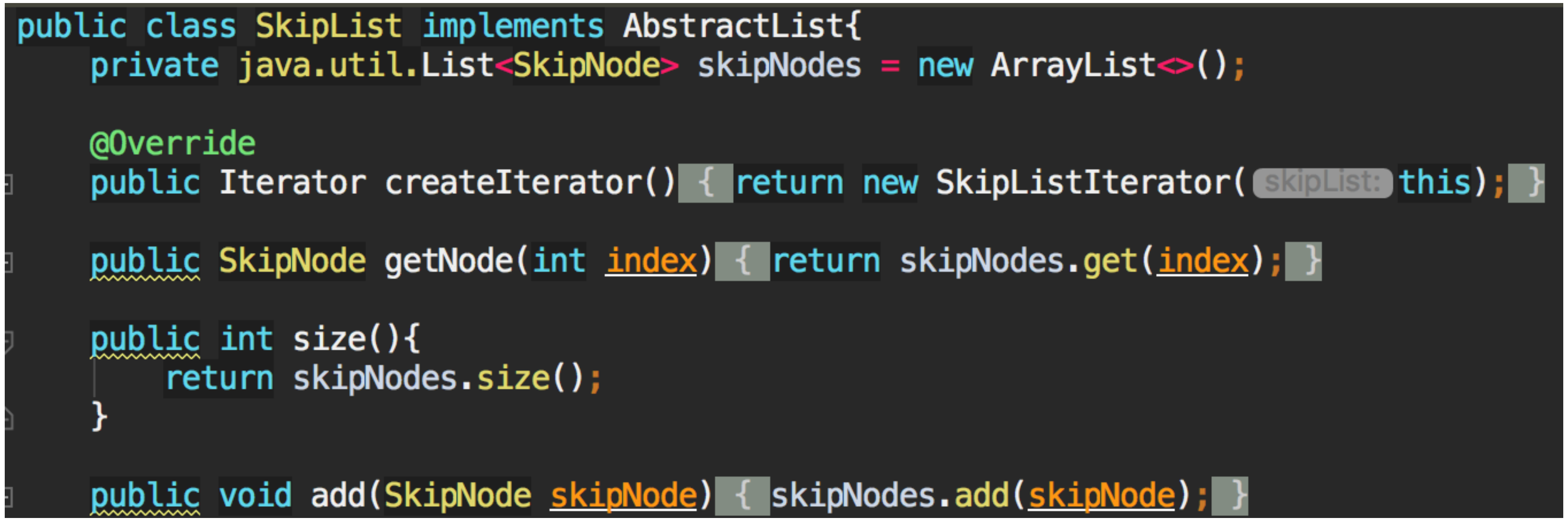
```
public interface AbstractList {
    public Iterator createIterator();

}
```

List



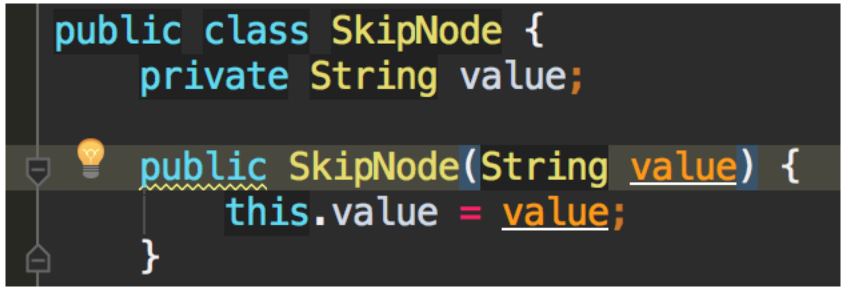
```
public class List implements AbstractList{
    public int length = 0;
    private String[] datas = new String[10000];

    @Override
    public Iterator createIterator() { return new ListIterator( list: this); }

    public String get(int index){
        if(index >= length ){
            return null;
        }
        else {
            return datas[index];
        }
    }

    public void add(String data){
        datas[length] = data;
        length ++;

        if(length == datas.length){
            datas = Arrays.copyOf(datas,  newLength: length * 2);
        }
    }
```

SkipList



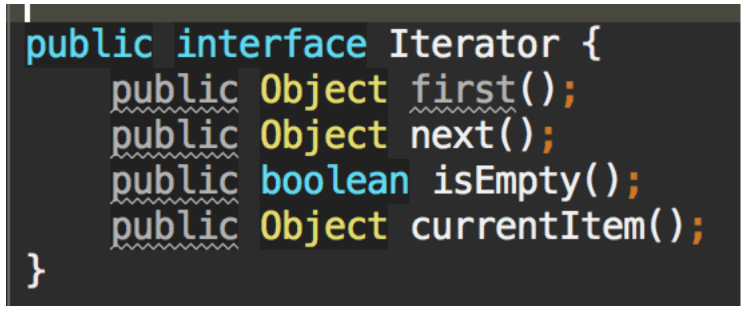
```
public class SkipList implements AbstractList{
    private java.util.List<SkipNode> skipNodes = new ArrayList<>();

    @Override
    public Iterator createIterator() { return new SkipListIterator( skipList: this); }

    public SkipNode getNode(int index) { return skipNodes.get(index); }

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

    public void add(SkipNode skipNode) { skipNodes.add(skipNode); }
```

SkipNode



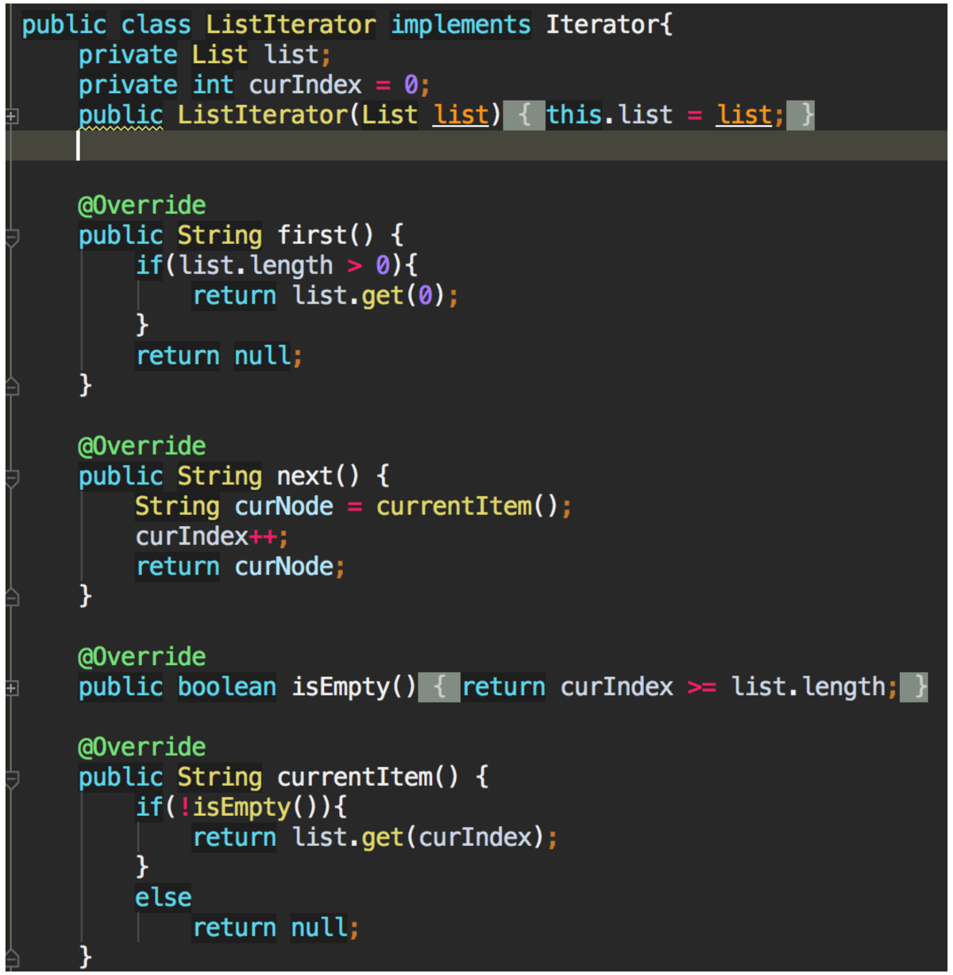
```
public class SkipNode {
    private String value;

    public SkipNode(String value) {
        this.value = value;
    }
```

Iterator

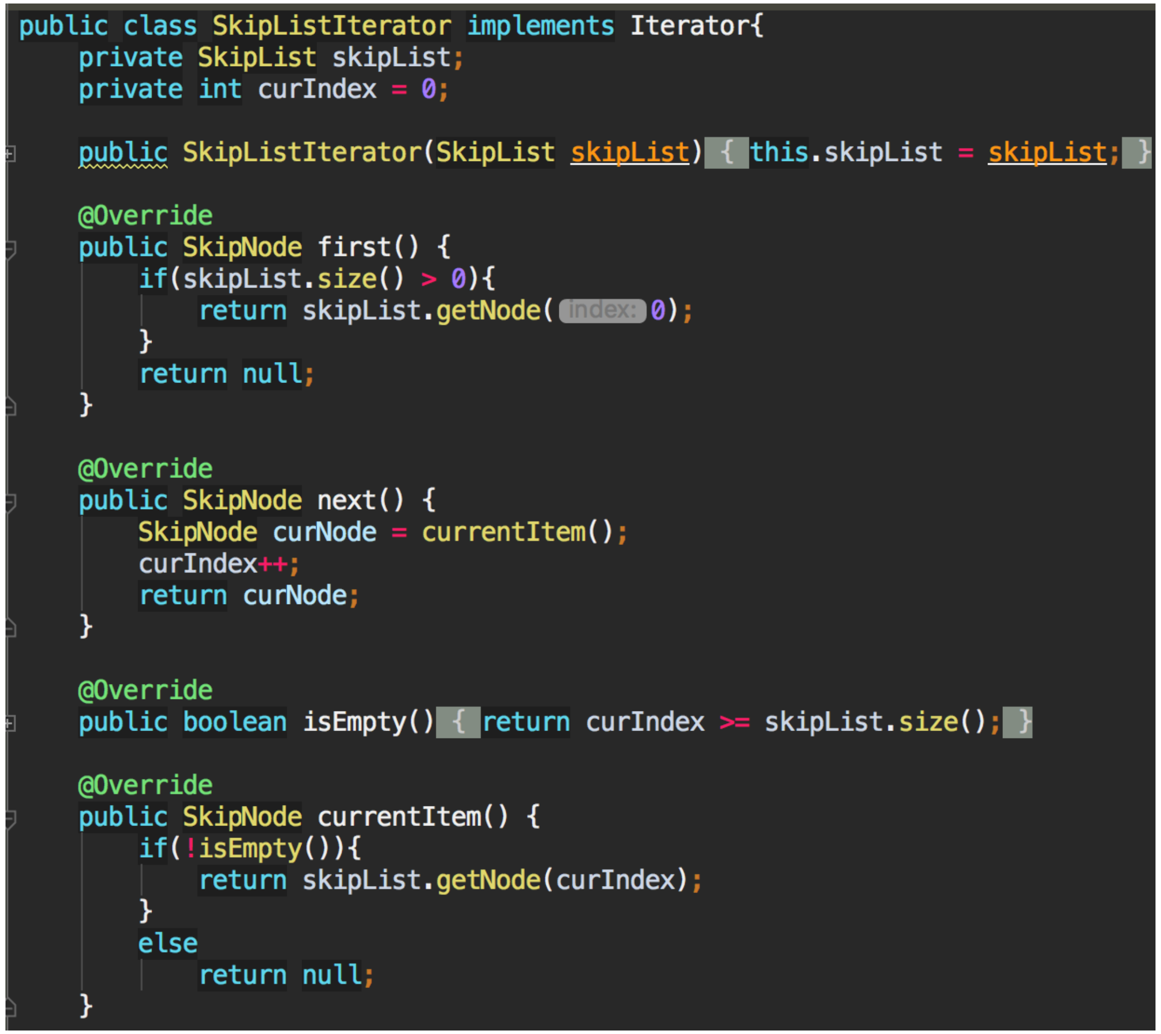


```
public interface Iterator {
    public Object first();
    public Object next();
    public boolean isEmpty();
    public Object currentItem();
}
```

ListIterator

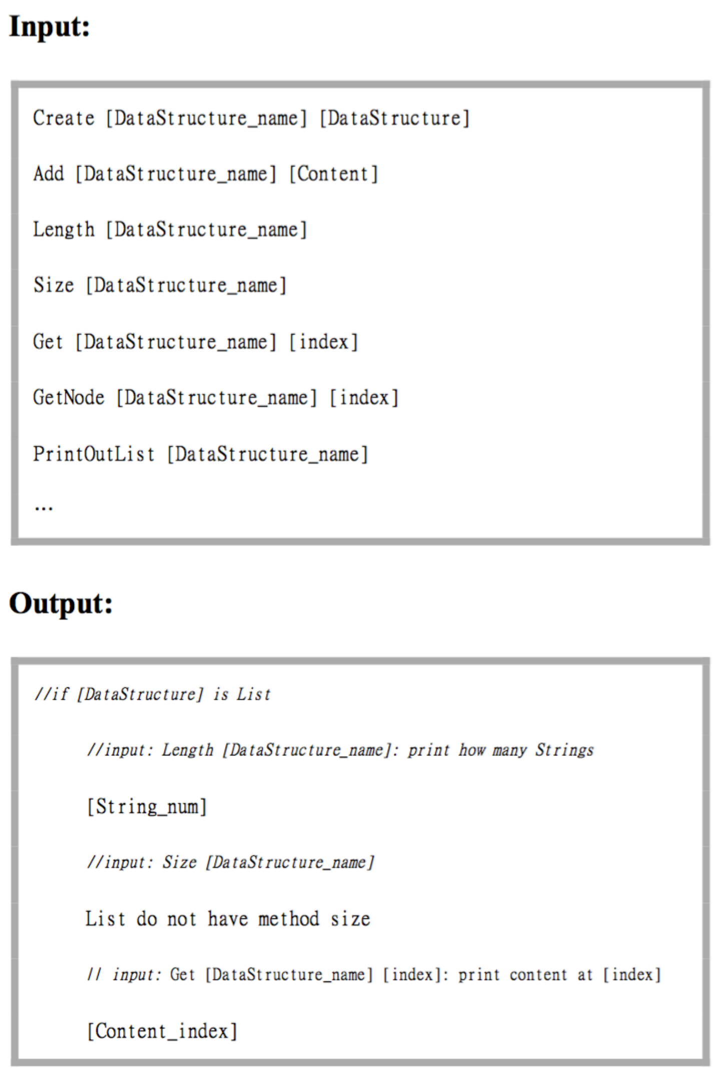

```
public class ListIterator implements Iterator{
    private List list;
    private int curIndex = 0;
    public ListIterator(List list) { this.list = list; }

    @Override
    public String first() {
        if(list.length > 0){
            return list.get(0);
        }
        return null;
    }

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

    @Override
    public boolean isEmpty() { return curIndex >= list.length; }

    @Override
    public String currentItem() {
        if(!isEmpty()){
            return list.get(curIndex);
        }
        else
            return null;
    }
```

SkipIterator



```
public class SkipListIterator implements Iterator{
    private SkipList skipList;
    private int curIndex = 0;

    public SkipListIterator(SkipList skipList) { this.skipList = skipList; }

    @Override
    public SkipNode first() {
        if(skipList.size() > 0){
            return skipList.getNode( index: 0);
        }
        return null;
    }

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

    @Override
    public boolean isEmpty() { return curIndex >= skipList.size(); }

    @Override
    public SkipNode currentItem() {
        if(!isEmpty()){
            return skipList.getNode(curIndex);
        }
        else
            return null;
    }
```

Input / Output



Input:

```
Create [DataStructure_name] [DataStructure]

Add [DataStructure_name] [Content]

Length [DataStructure_name]

Size [DataStructure_name]

Get [DataStructure_name] [index]

GetNode [DataStructure_name] [index]

PrintOutList [DataStructure_name]

...
```

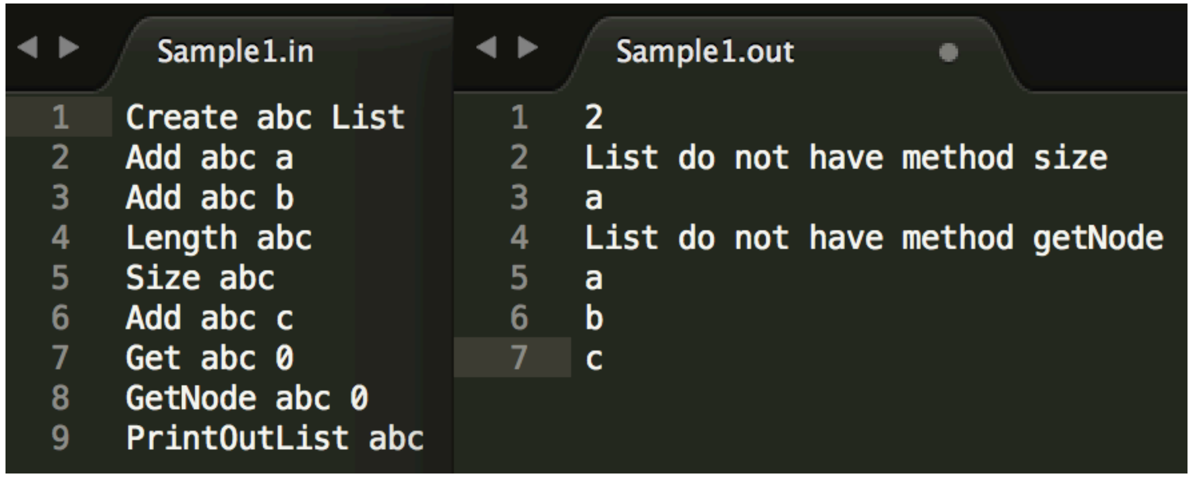
Output:

```
//if [DataStructure] is List

    //input: Length [DataStructure_name]: print how many Strings

    [String_num]

    //input: Size [DataStructure_name]

    List do not have method size

    // input: Get [DataStructure_name] [index]: print content at [index]

    [Content_index]
```

Test cases

- ❑ TestCase 1: List (Include Invalid Command Size and GetNode)
- ❑ TestCase 2: SkipList (Include Invalid Command Length and Get)
- ❑ TestCase 3: Complex

Test case1

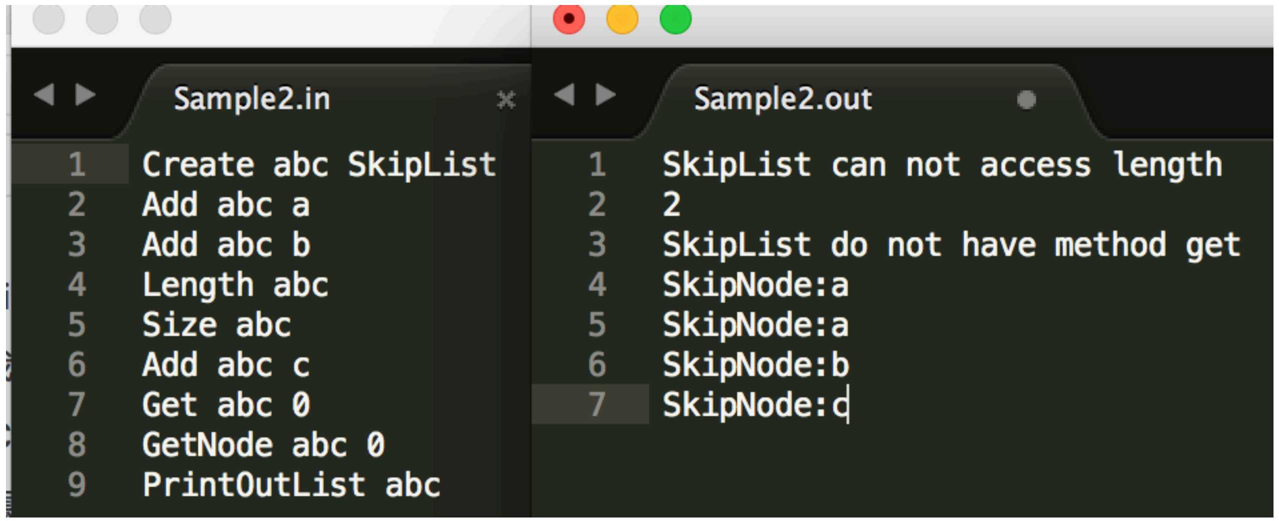

Sample1.in

```
Create abc List
Add abc a
Add abc b
Length abc
Size abc
Add abc c
Get abc 0
GetNode abc 0
PrintOutList abc
```

Sample1.out

```
2
List do not have method size
a
List do not have method getNode
a
b
c
```

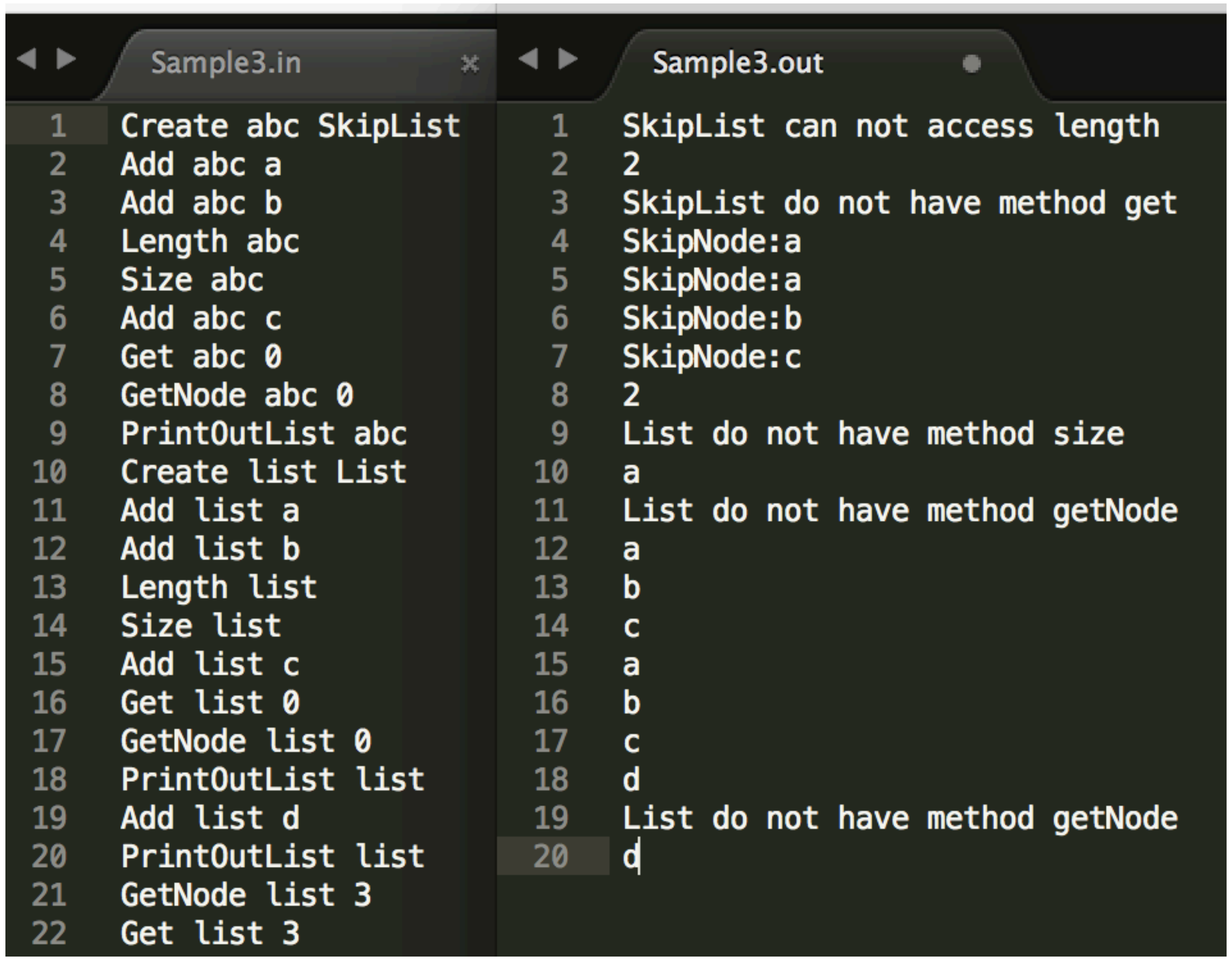
Test case2



```
Create abc SkipList
Add abc a
Add abc b
Length abc
Size abc
Add abc c
Get abc 0
GetNode abc 0
PrintOutList abc
```

```
SkipList can not access length
2
SkipList do not have method get
SkipNode:a
SkipNode:a
SkipNode:b
SkipNode:c
```

Test case3



Sample3.in

```
Create abc SkipList
Add abc a
Add abc b
Length abc
Size abc
Add abc c
Get abc 0
GetNode abc 0
PrintOutList abc
Create list List
Add list a
Add list b
Length list
Size list
Add list c
Get list 0
GetNode list 0
PrintOutList list
Add list d
PrintOutList list
GetNode list 3
Get list 3
```

Sample3.out

```
SkipList can not access length
2
SkipList do not have method get
SkipNode:a
SkipNode:a
SkipNode:b
SkipNode:c
2
List do not have method size
a
List do not have method getNode
a
b
c
a
b
c
d
List do not have method getNode
d
```

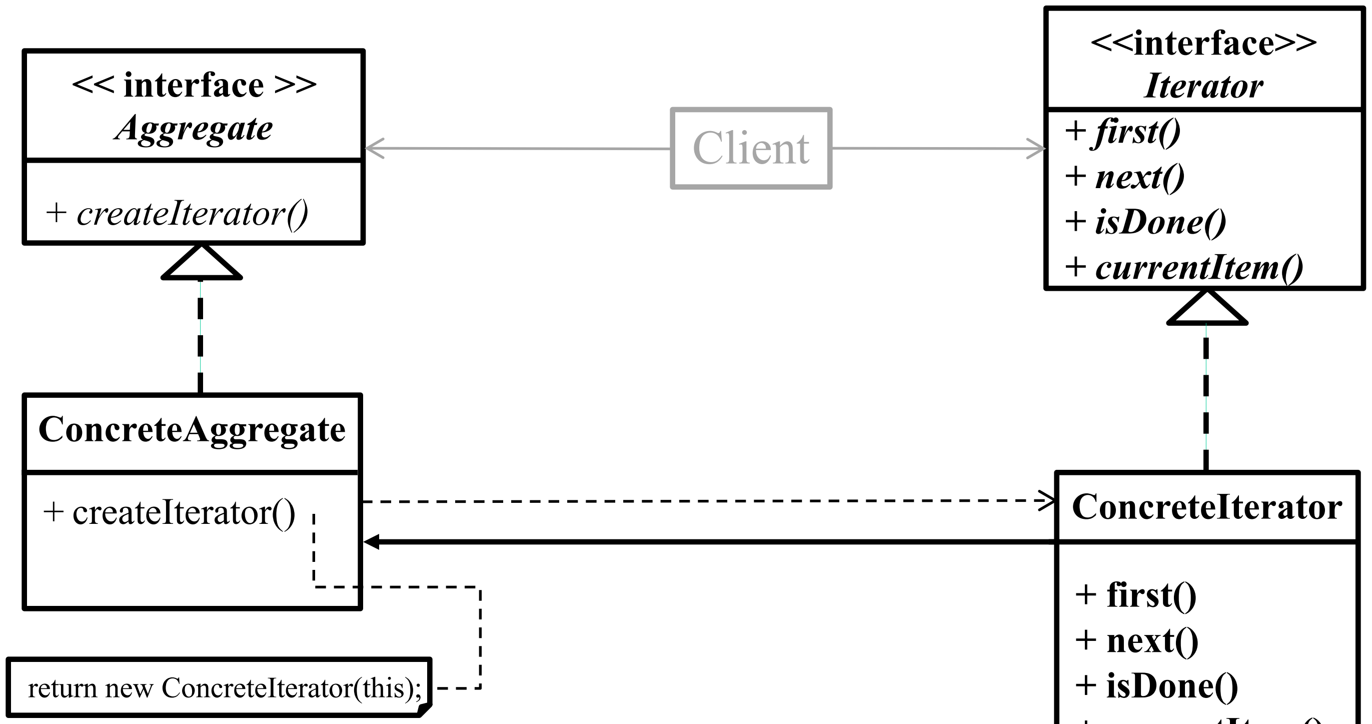
Recurrent Problem

- The method of accessing the elements of two aggregate objects with different representations will be modified if a new aggregate object with different representation is added.
  - An aggregate object such as a list gives you a way to access its elements without exposing its internal structure.
  - Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don't want to bloat the List interface with operations for different traversals, even if you can anticipate the ones you will need.
  - You may also need to have more than one traversal pending on the same list.

Intent

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator Pattern Structure$_1$



Iterator Pattern Structure$_2$

1. Client creates a ConcreteAggregate.

2. Client invokes createIterator() to get a ConcreteIterator.

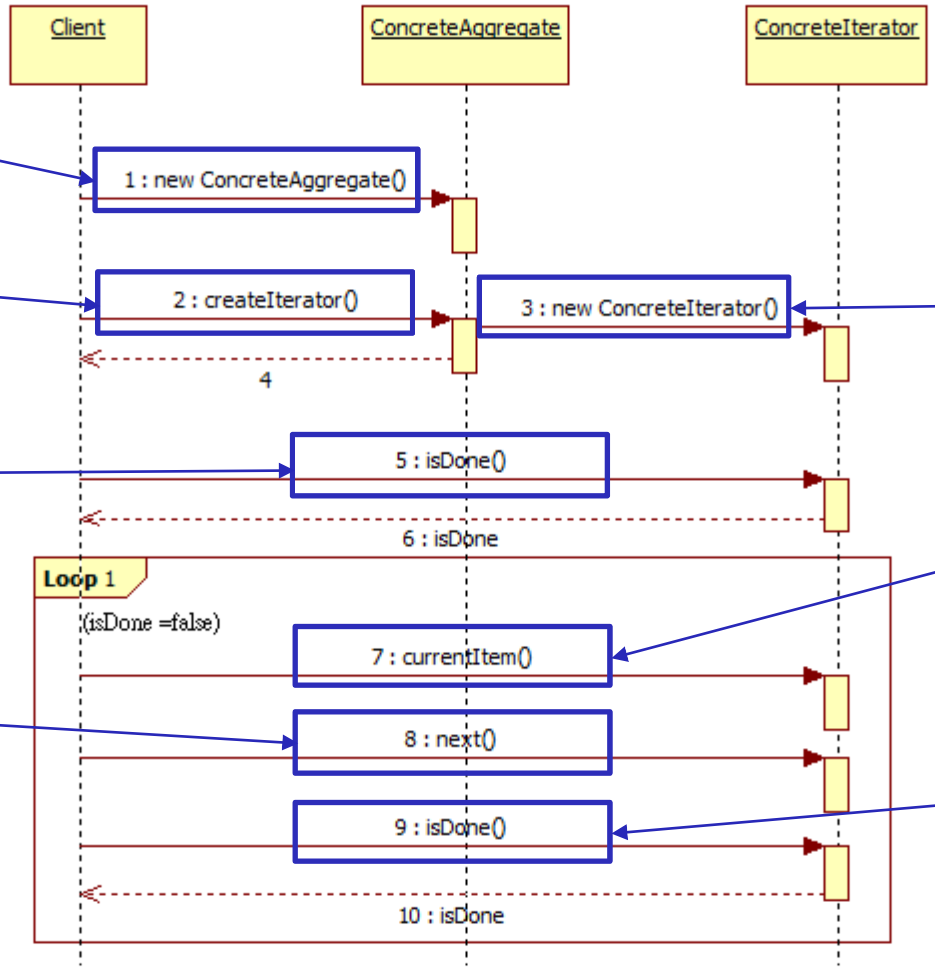
3. ConcreteAggregate creates a ConcreteIterator.

4. Client invokes isDone() to test whether we've advanced beyond the last element

5. Client invokes currentItem() to return the current element in the list.

6. Client invokes next() to advance the current element to the next element.

7. Client invokes isDone() again to test whether we've advanced beyond the last element

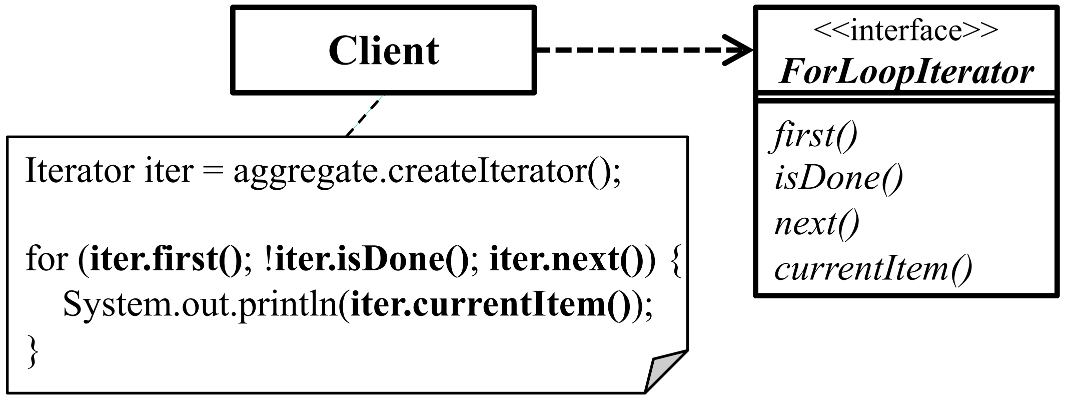


Iterator Pattern Structure$_3$

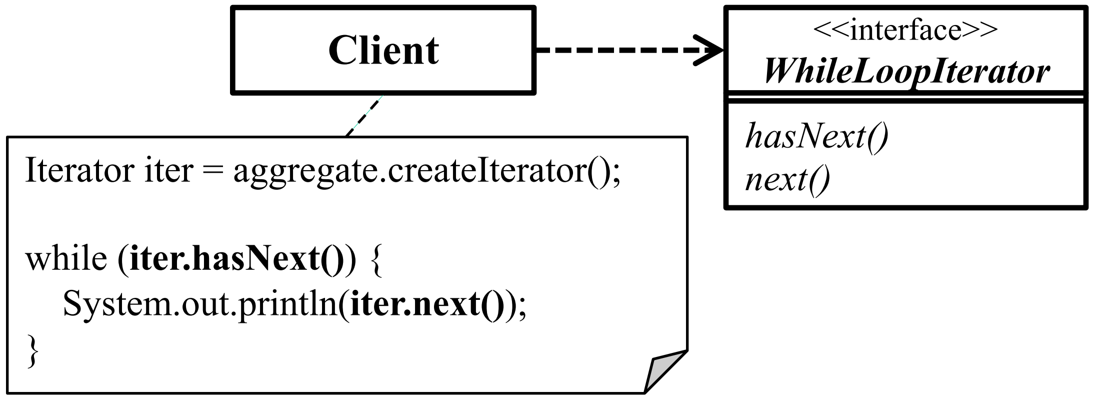
	Instantiation	Use	Termination
Client	Other class except classes in the Iterator Pattern	Other class except classes in the Iterator Pattern	Other class except classes in the Iterator Pattern
Aggregate	X	Client class uses this interface to get a ConcreteIterator through polymorphism	X
Concrete Aggregate	Other class or the client class	Client class uses this class to get a ConcreteIterator through Aggregate	Other class or the client class
Iterator	X	Client class uses ConcreteIterator through this interface	X
Concrete Iterator	ConcreteAggregate	Client class use this class to access the elements of an aggregate object sequentially	Other class or the client class

Two kinds of Iterator

- Two kinds of Iterators for different iterations
  - For-Loop
    - **first()**: rewind to the first element in the aggregate.
    - **isDone()**: check if all elements are traversed.
    - **next()**: advance to the next element.
    - **currentItem()**: return the current element.
  - While-Loop
    - **hasNext()**: check if there is any element that has not been traversed.
    - **next()**: advance to the next element and return it.



```
Iterator iter = aggregate.createIterator();

for (iter.first(); !iter.isDone(); iter.next()) {
   System.out.println(iter.currentItem());
}
```



```
Iterator iter = aggregate.createIterator();

while (iter.hasNext()) {
   System.out.println(iter.next());
}
```

Merge Two Menus

Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering

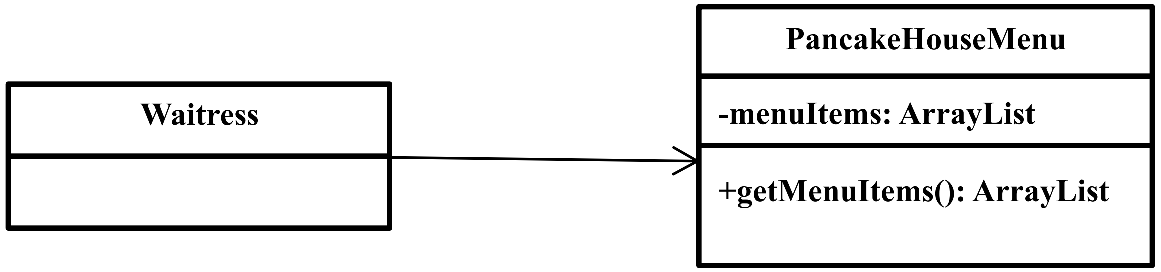
National Taiwan University

Requirements Statement

- 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 services 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.

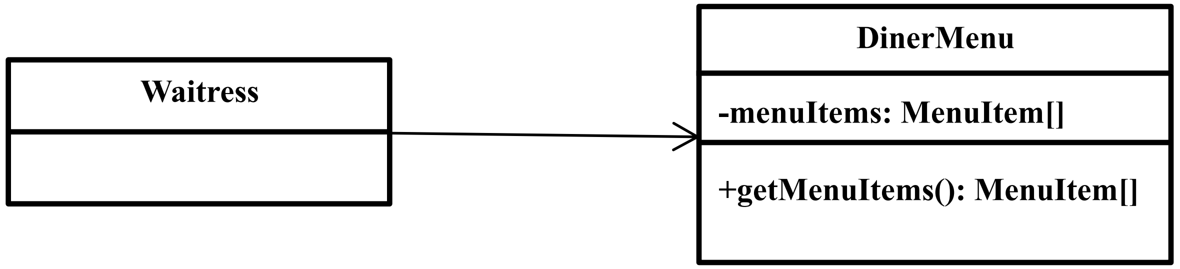
Requirements Statement

- A waitress of Pancake House keeps a breakfast menu which uses an ArrayList to hold its menu items.



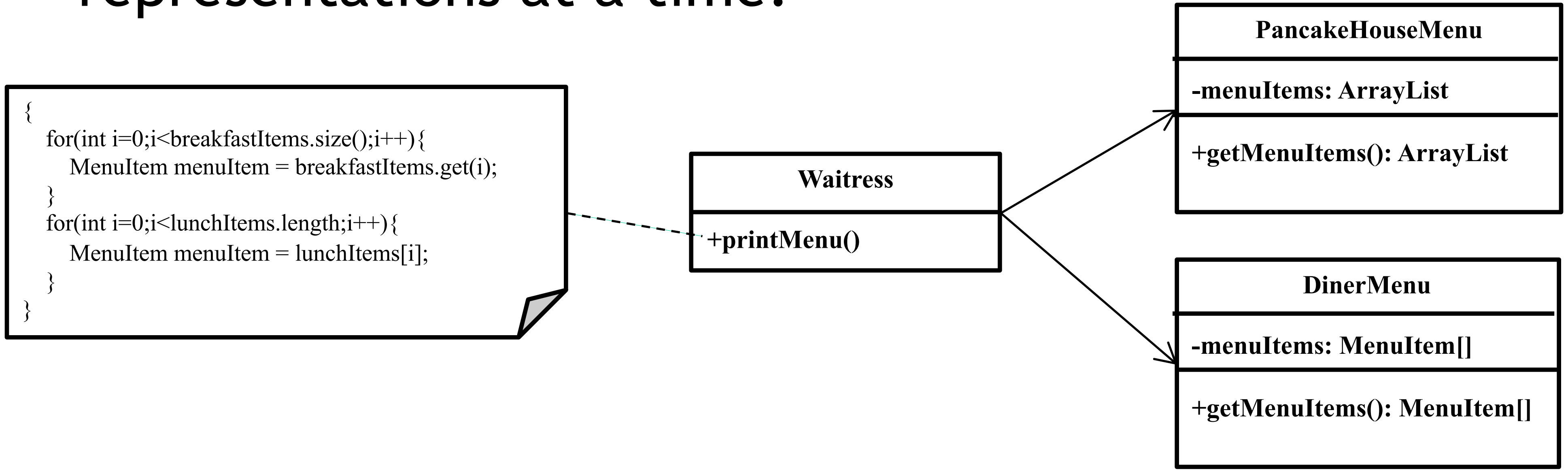
Requirements Statement

- And a waitress of Diner keeps a lunch menu which uses an array to hold its menu items.



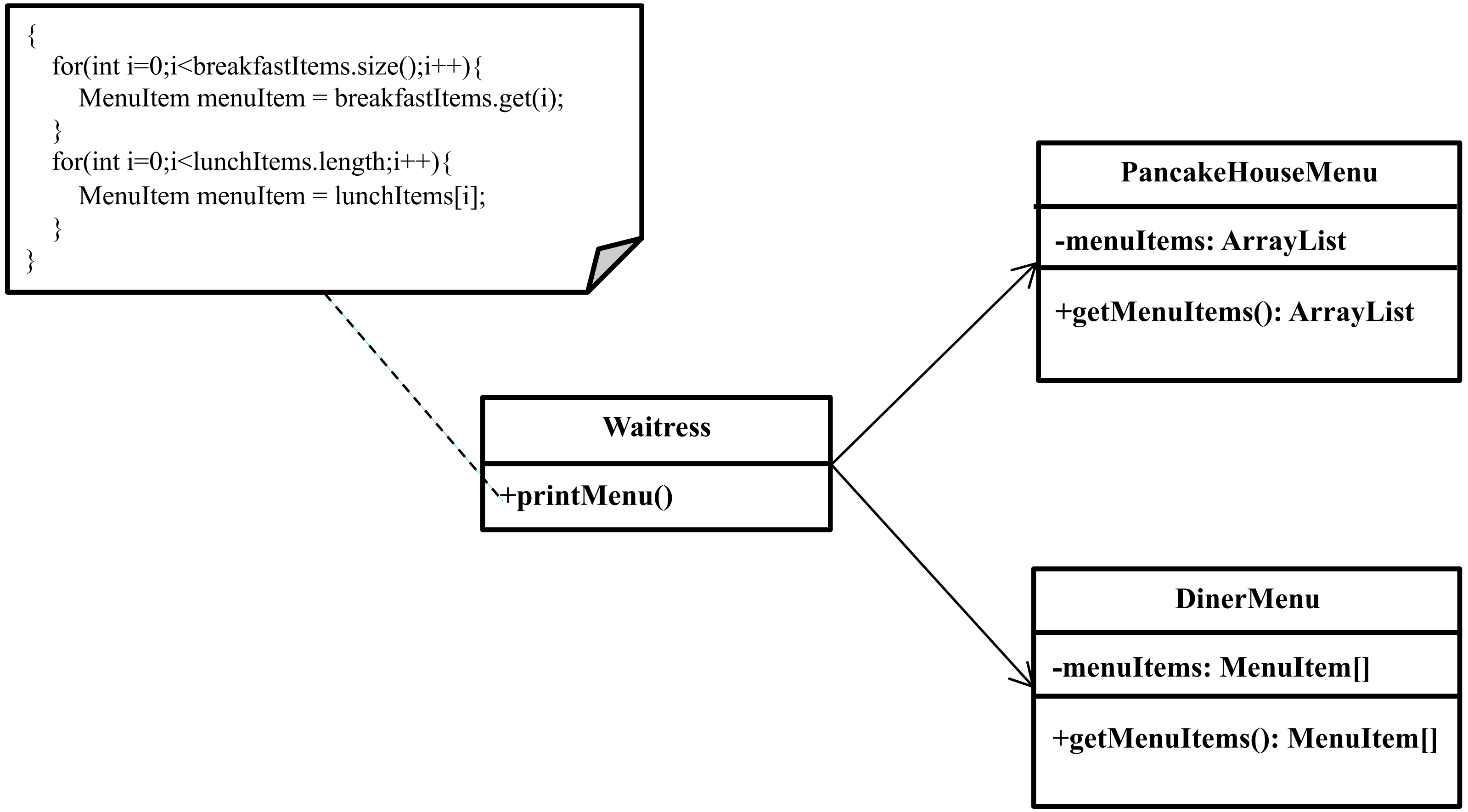
Requirements Statement

- 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.

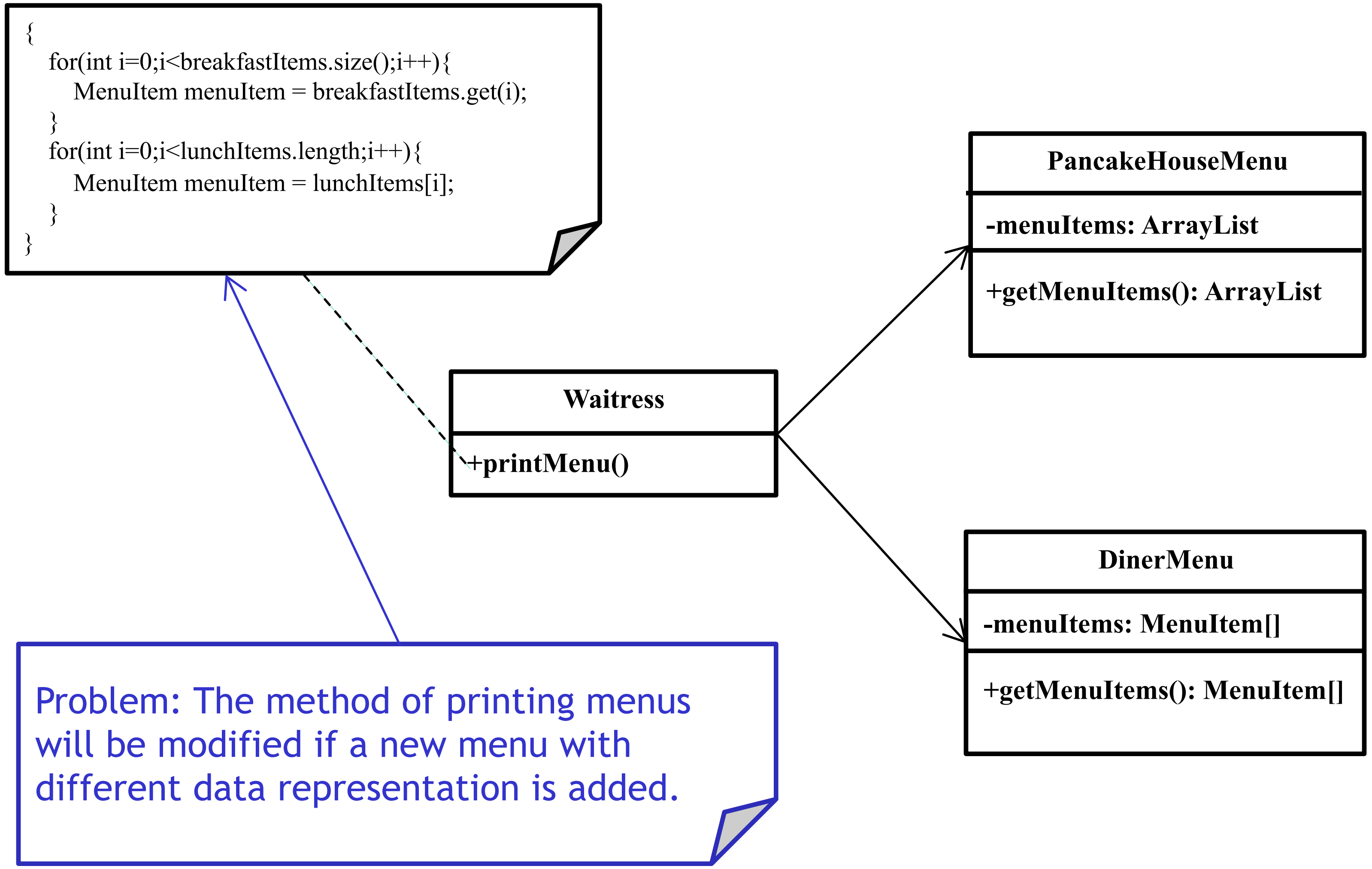


```
{
   for(int i=0;i<breakfastItems.size();i++){
      MenuItem menuItem = breakfastItems.get(i);
   }
   for(int i=0;i<lunchItems.length;i++){
      MenuItem menuItem = lunchItems[i];
   }
}
```

Initial Design - Class Diagram


```
{
   for(int i=0;i<breakfastItems.size();i++){
      MenuItem menuItem = breakfastItems.get(i);
   }
   for(int i=0;i<lunchItems.length;i++){
      MenuItem menuItem = lunchItems[i];
   }
}
```

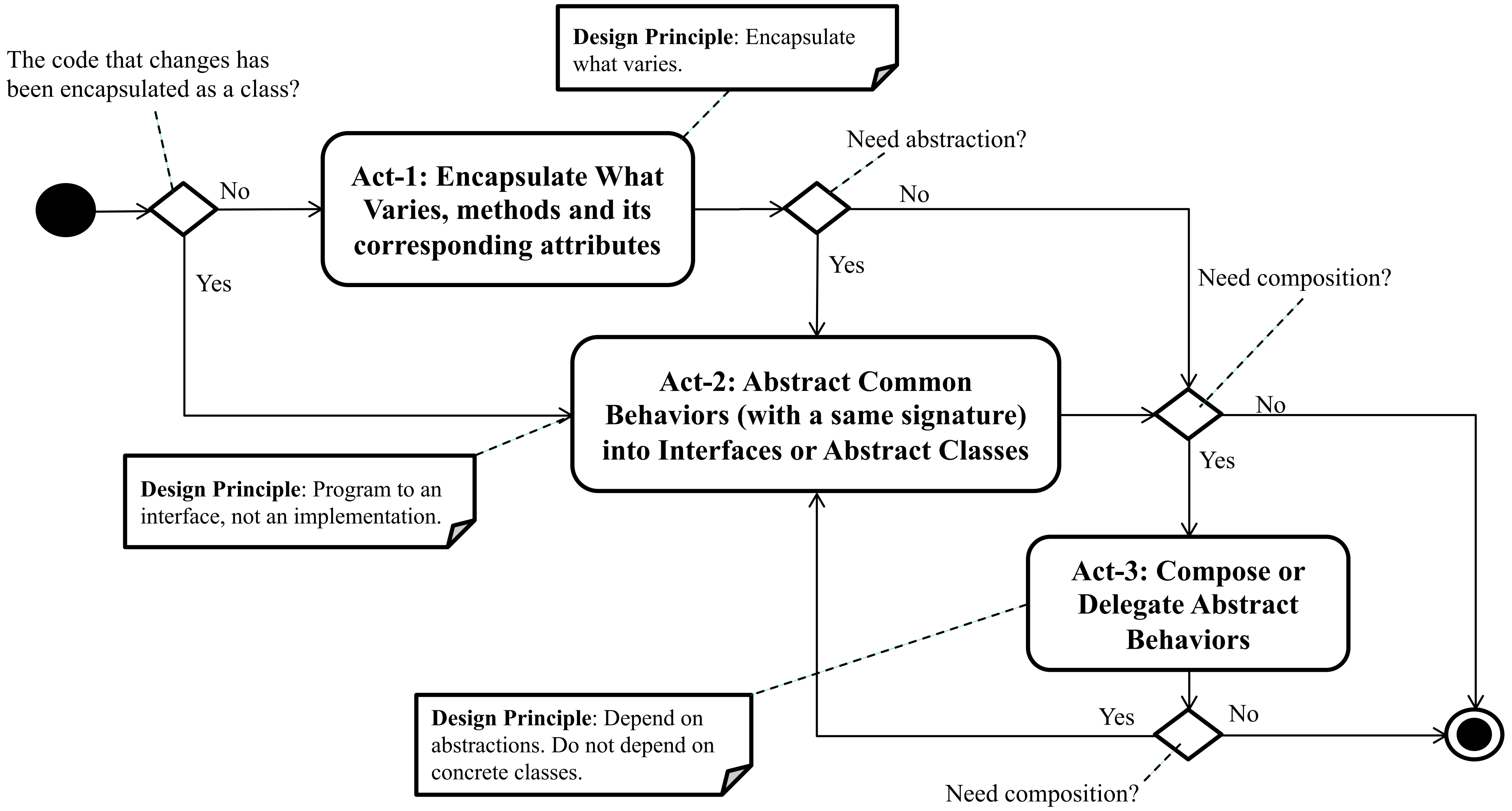
Problems with Initial Design



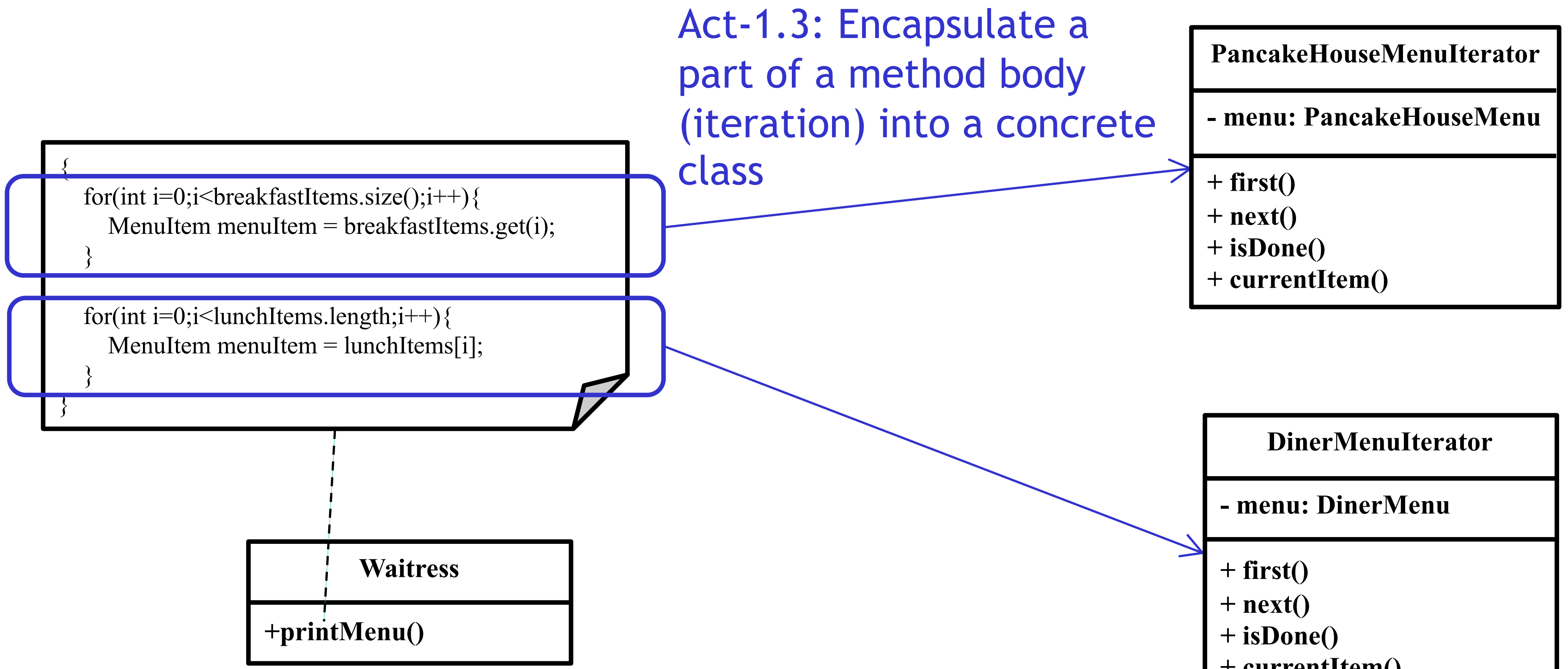
```
{
   for(int i=0;i<breakfastItems.size();i++){
      MenuItem menuItem = breakfastItems.get(i);
   }
   for(int i=0;i<lunchItems.length;i++){
      MenuItem menuItem = lunchItems[i];
   }
}
```

Problem: The method of printing menus will be modified if a new menu with different data representation is added.

Design Process for Change

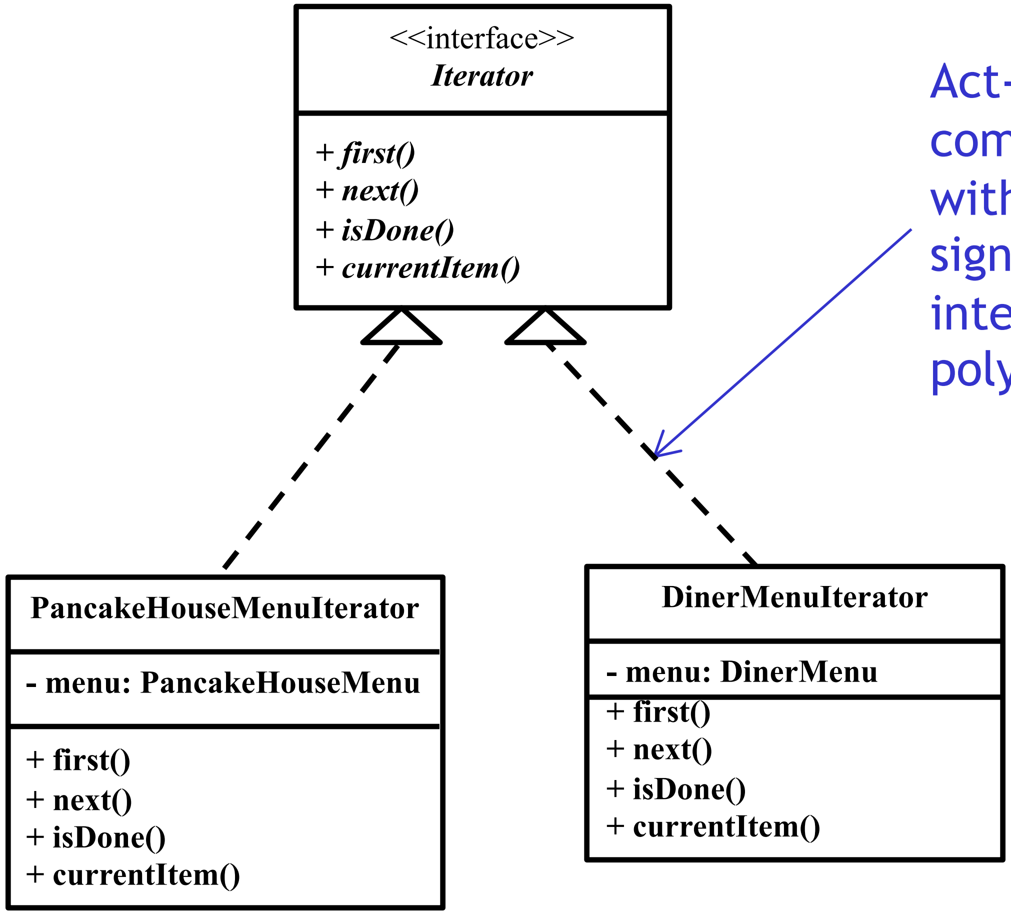

Design Principle: Encapsulate what varies.
The code that changes has been encapsulated as a class?
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract 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-1.3: Encapsulate What Varies



```
{
    for(int i=0;i<breakfastItems.size();i++){
        MenuItem menuItem = breakfastItems.get(i);
    }

    for(int i=0;i<lunchItems.length;i++){
        MenuItem menuItem = lunchItems[i];
    }
}
```

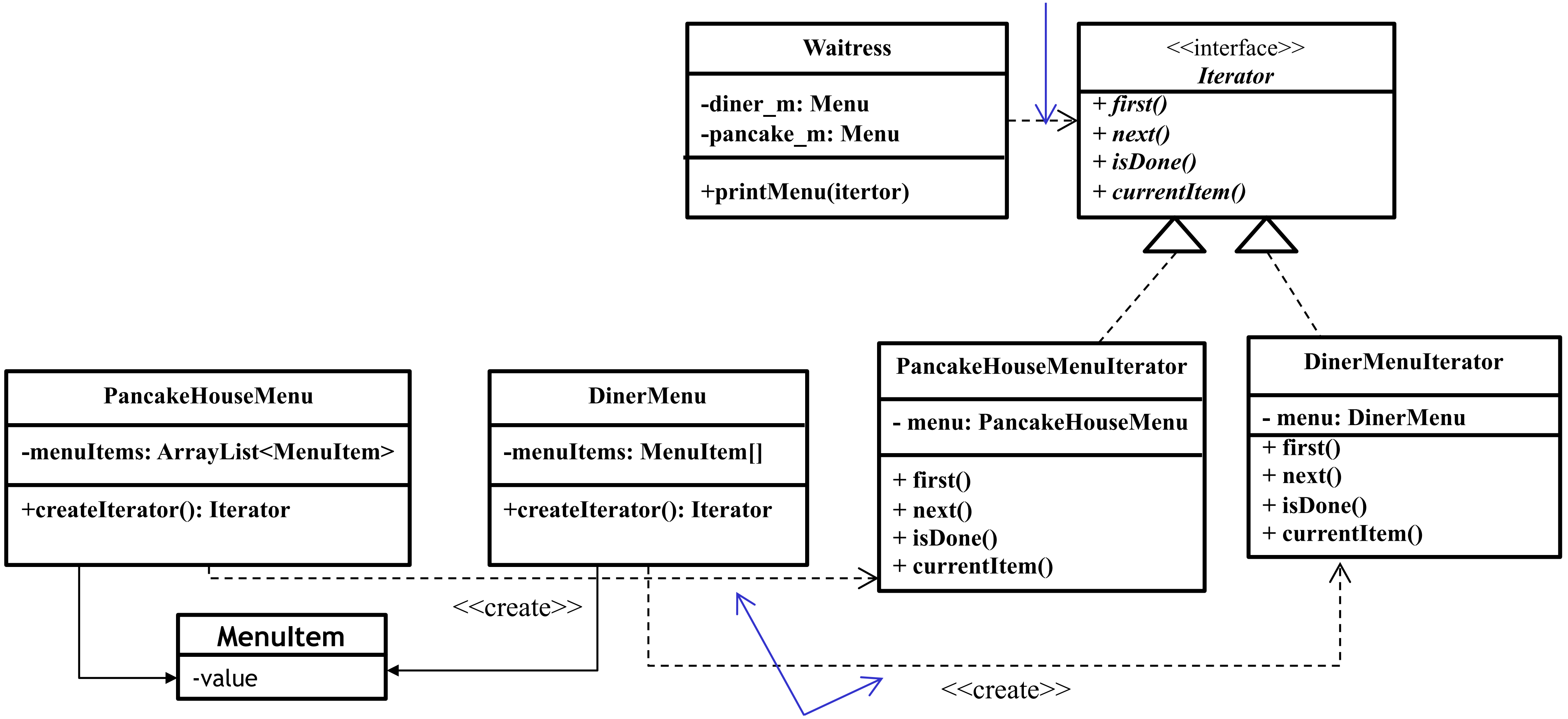
Act-2.1: Abstract Common Behaviors



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

Act-3: Delegate Behaviors

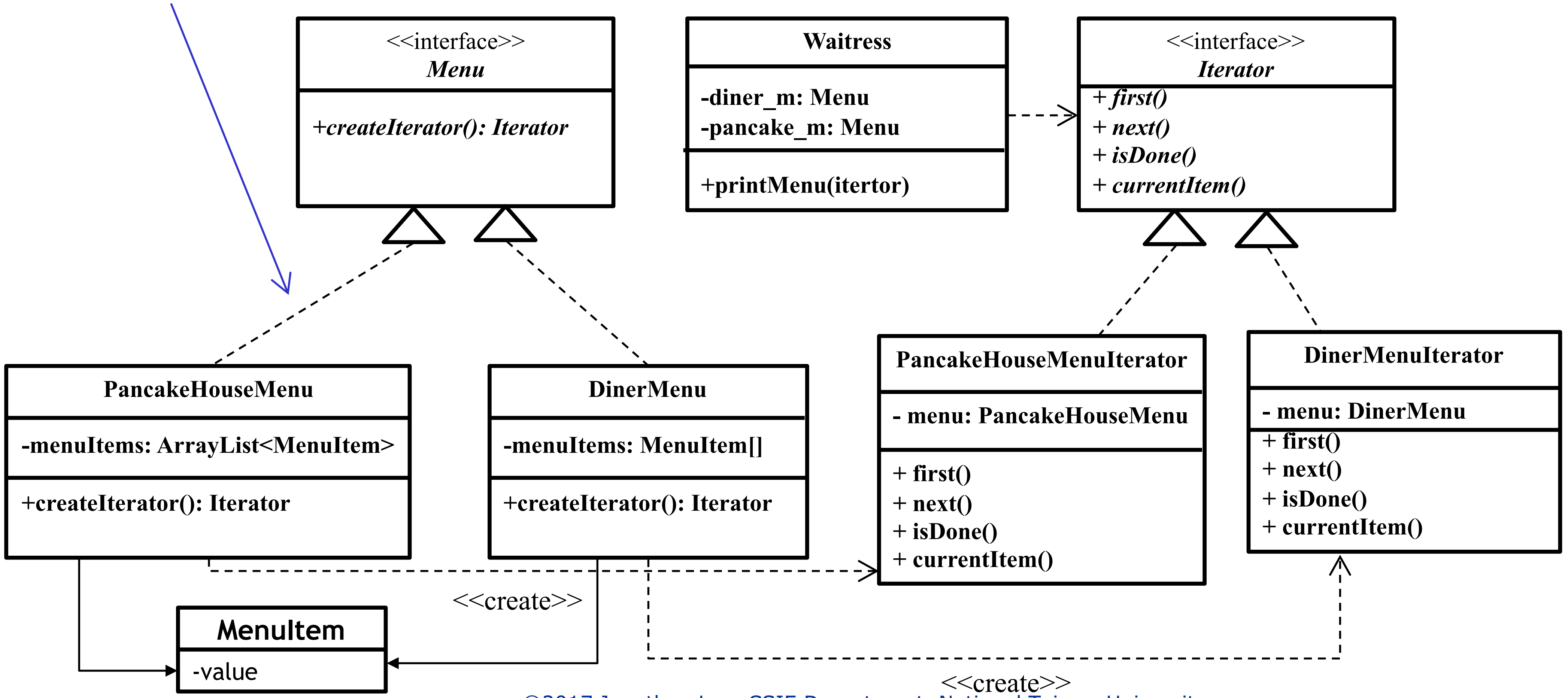
Act-3.3: Delegate behavior to an interface (by calling *next*() …)



Act-3.4: Delegate behavior to a concrete class (by calling the constructors of the iterator objects)

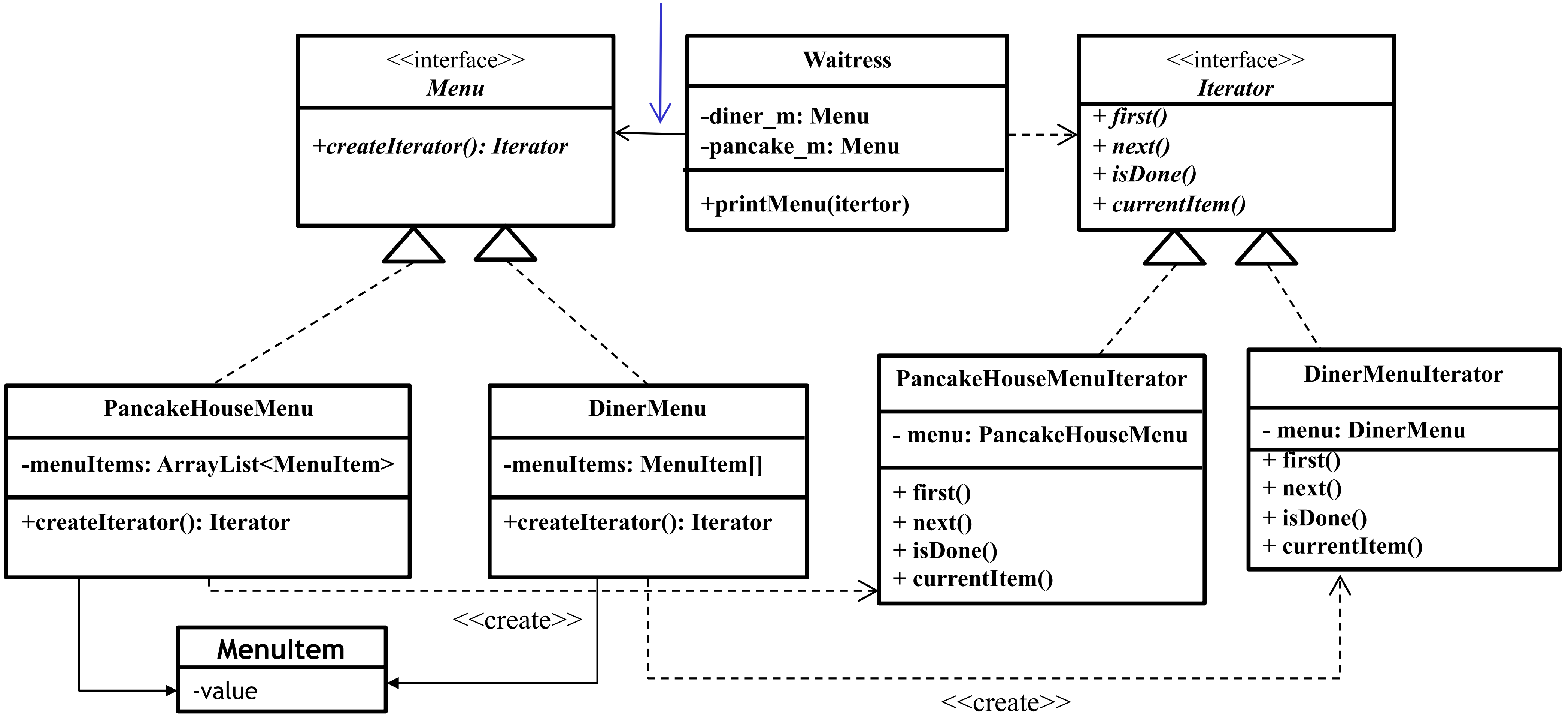
Act-2: Abstract Common Behaviors

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

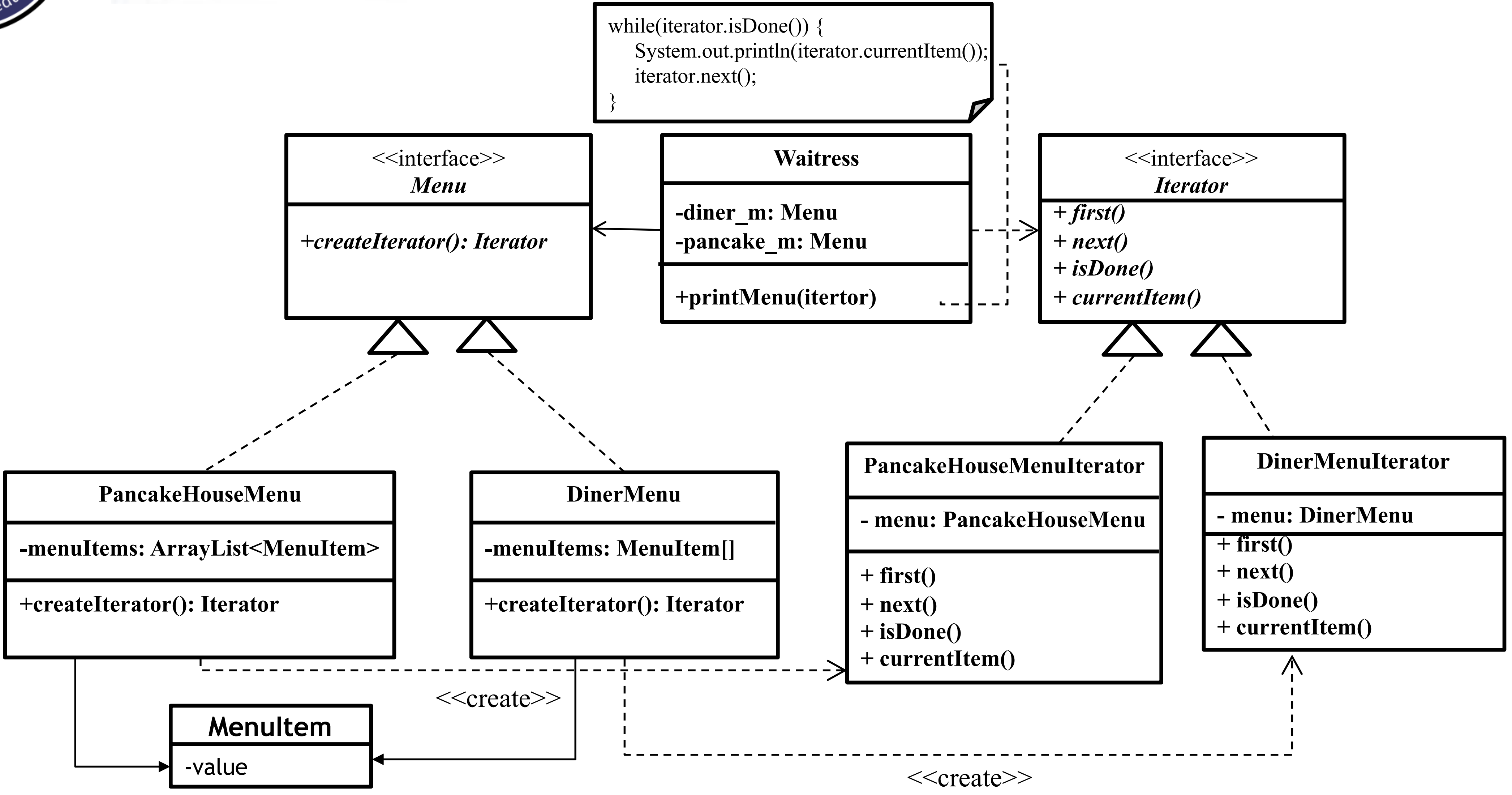


Act-3: Compose Abstract Interfaces/Abstract Classes

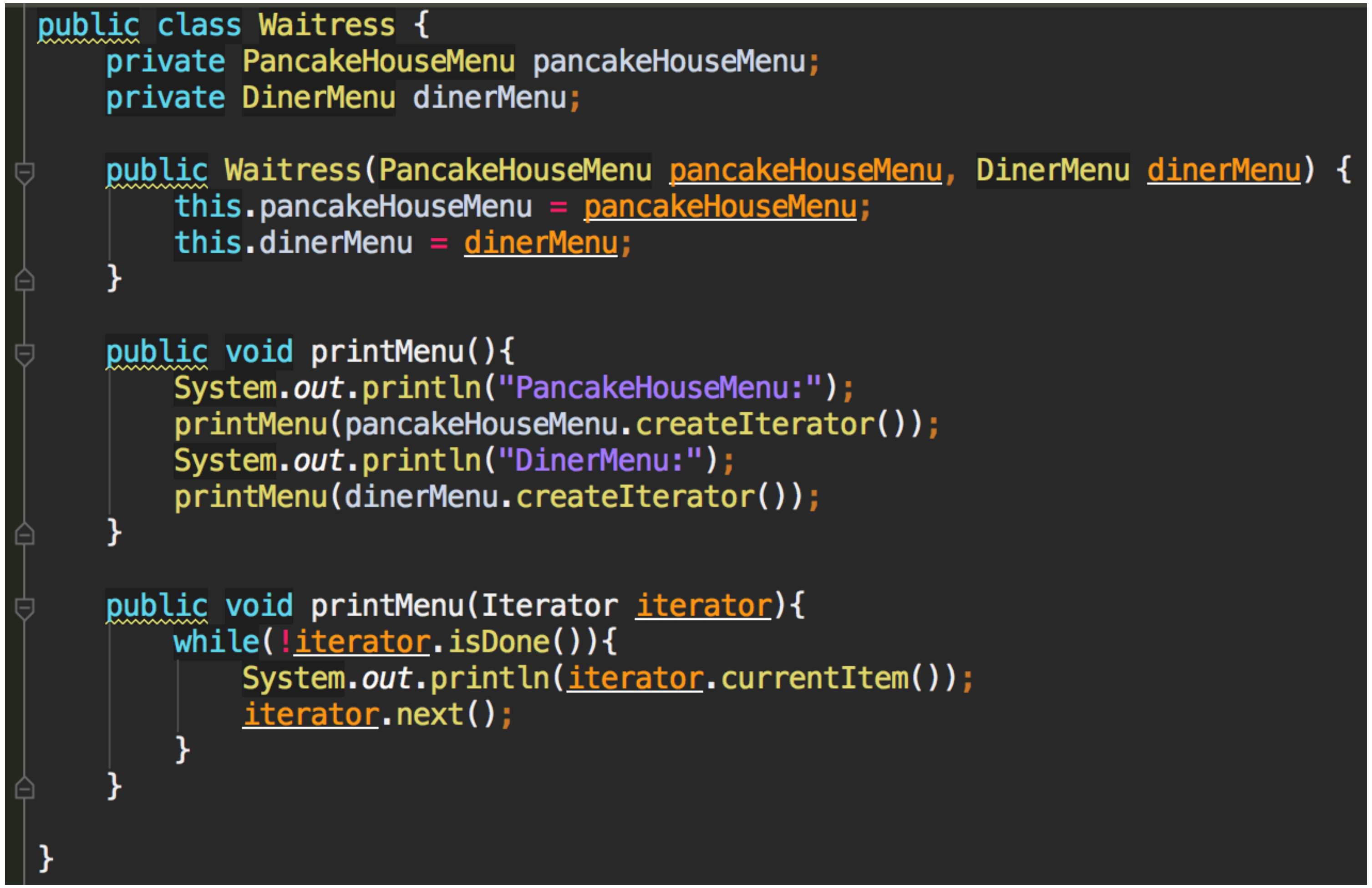
Act-3.1: Compose behaviors of an interface (through Waitress's attributes)



Refactored Design after Design Process

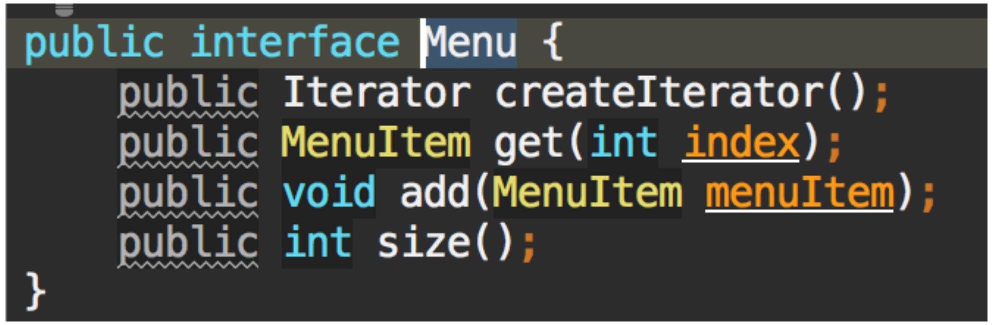


Waitress



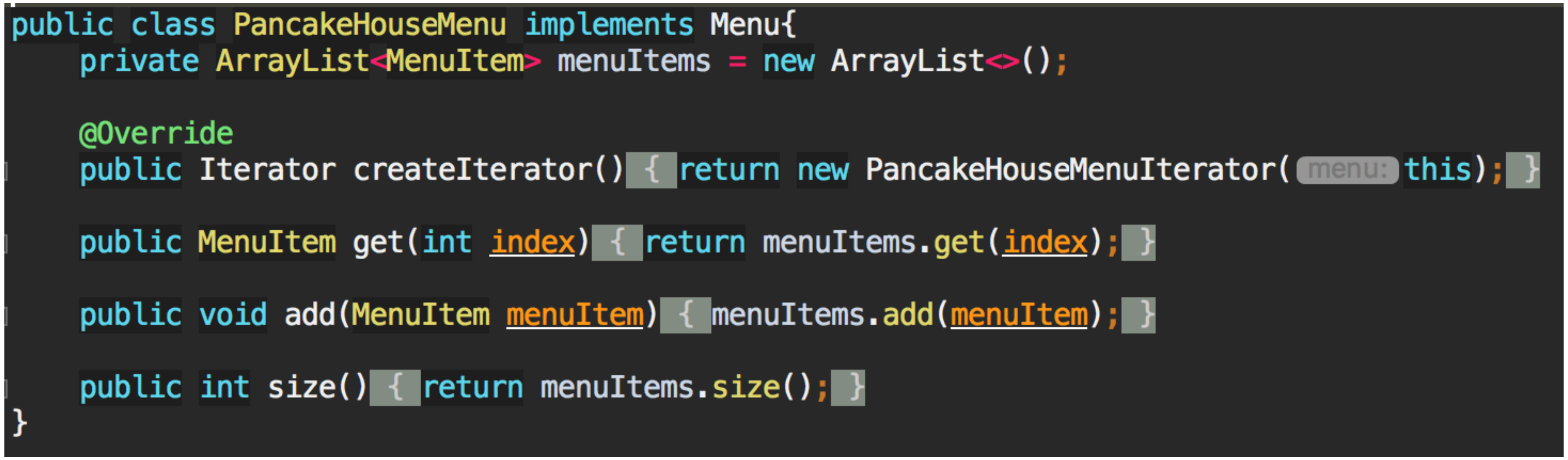
```java
public class Waitress {
    private PancakeHouseMenu pancakeHouseMenu;
    private DinerMenu dinerMenu;

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

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

    public void printMenu(Iterator iterator){
        while(!iterator.isDone()){
            System.out.println(iterator.currentItem());
            iterator.next();
        }
    }

}
```

Menu



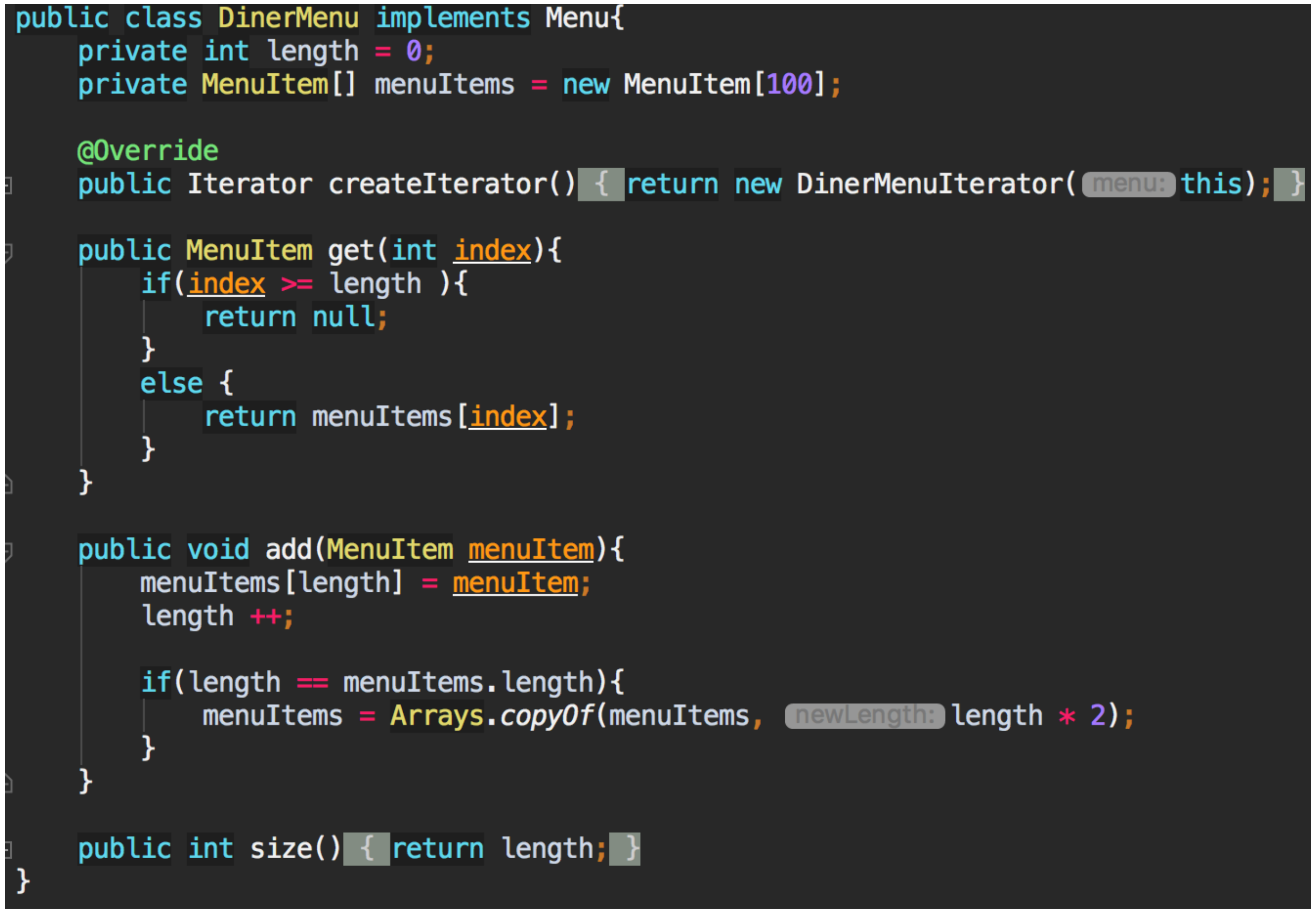
```
public interface Menu {
    public Iterator createIterator();
    public MenuItem get(int index);
    public void add(MenuItem menuItem);
    public int size();
}
```

PancakeHouseMenu



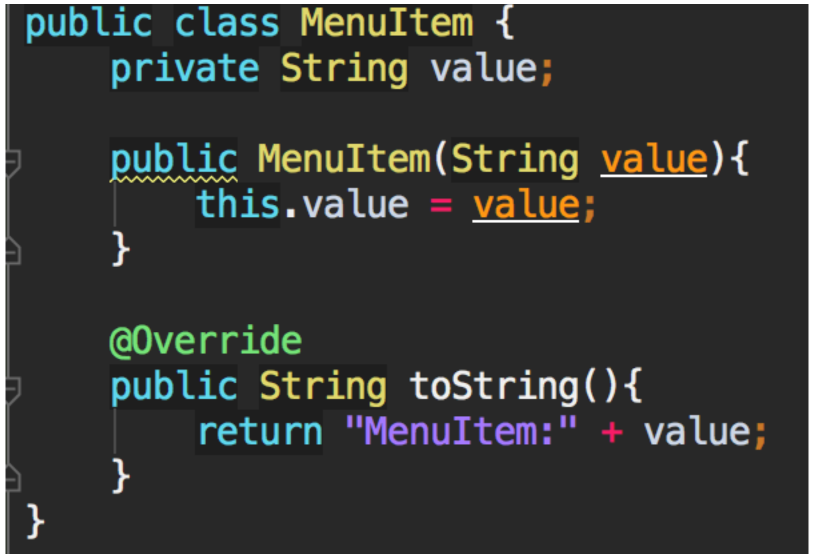
```java
public class PancakeHouseMenu implements Menu{
    private ArrayList<MenuItem> menuItems = new ArrayList<>();

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

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

    public void add(MenuItem menuItem) { menuItems.add(menuItem); }

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

DinerMenu



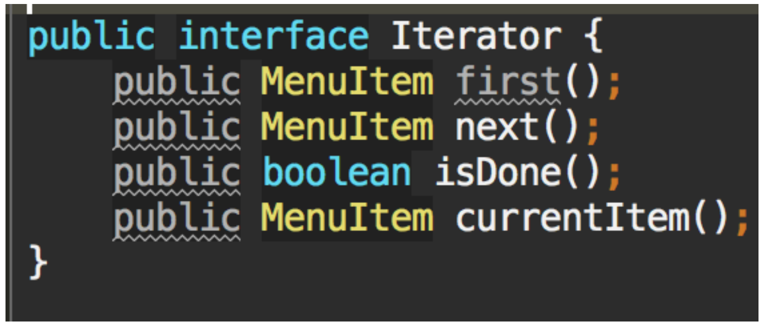
```java
public class DinerMenu implements Menu{
    private int length = 0;
    private MenuItem[] menuItems = new MenuItem[100];

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

    public MenuItem get(int index){
        if(index >= length ){
            return null;
        }
        else {
            return menuItems[index];
        }
    }

    public void add(MenuItem menuItem){
        menuItems[length] = menuItem;
        length ++;

        if(length == menuItems.length){
            menuItems = Arrays.copyOf(menuItems,  newLength: length * 2);
        }
    }

    public int size() { return length; }
}
```

MenuItem



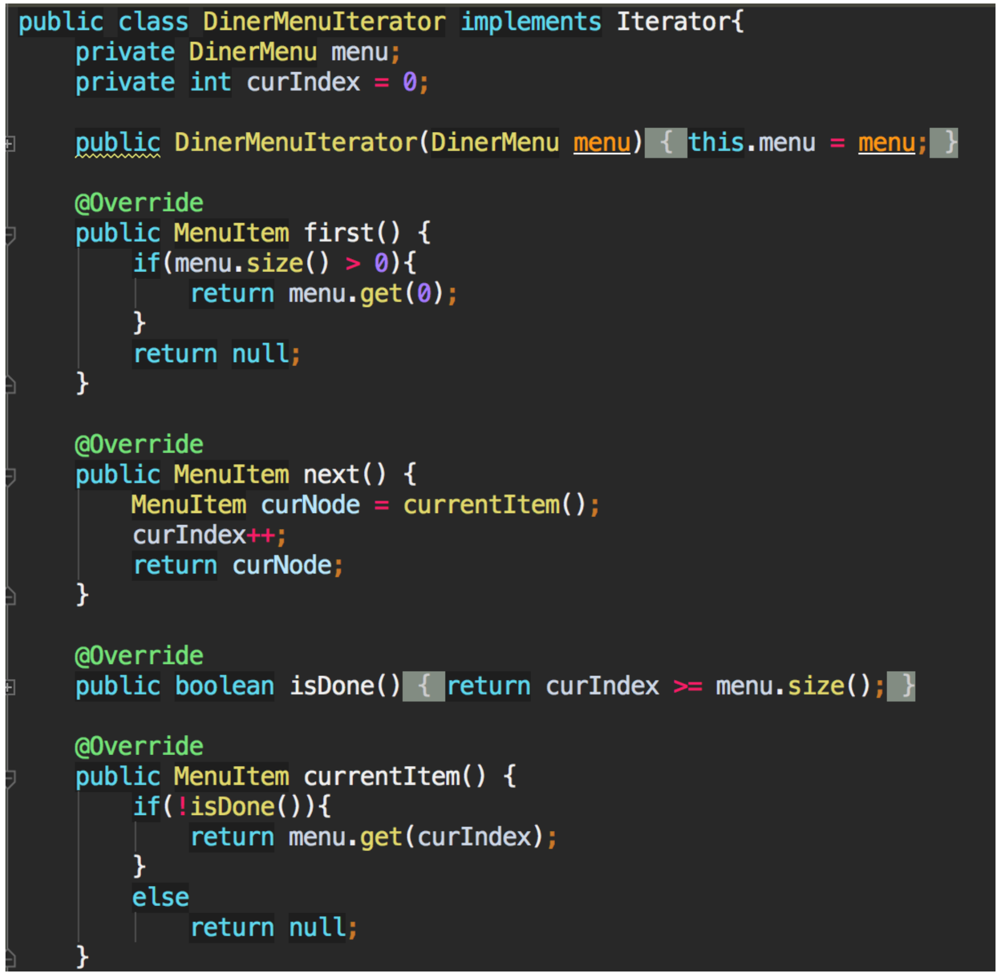
```
public class MenuItem {
    private String value;

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

    @Override
    public String toString(){
        return "MenuItem:" + value;
    }
}
```

Iterator



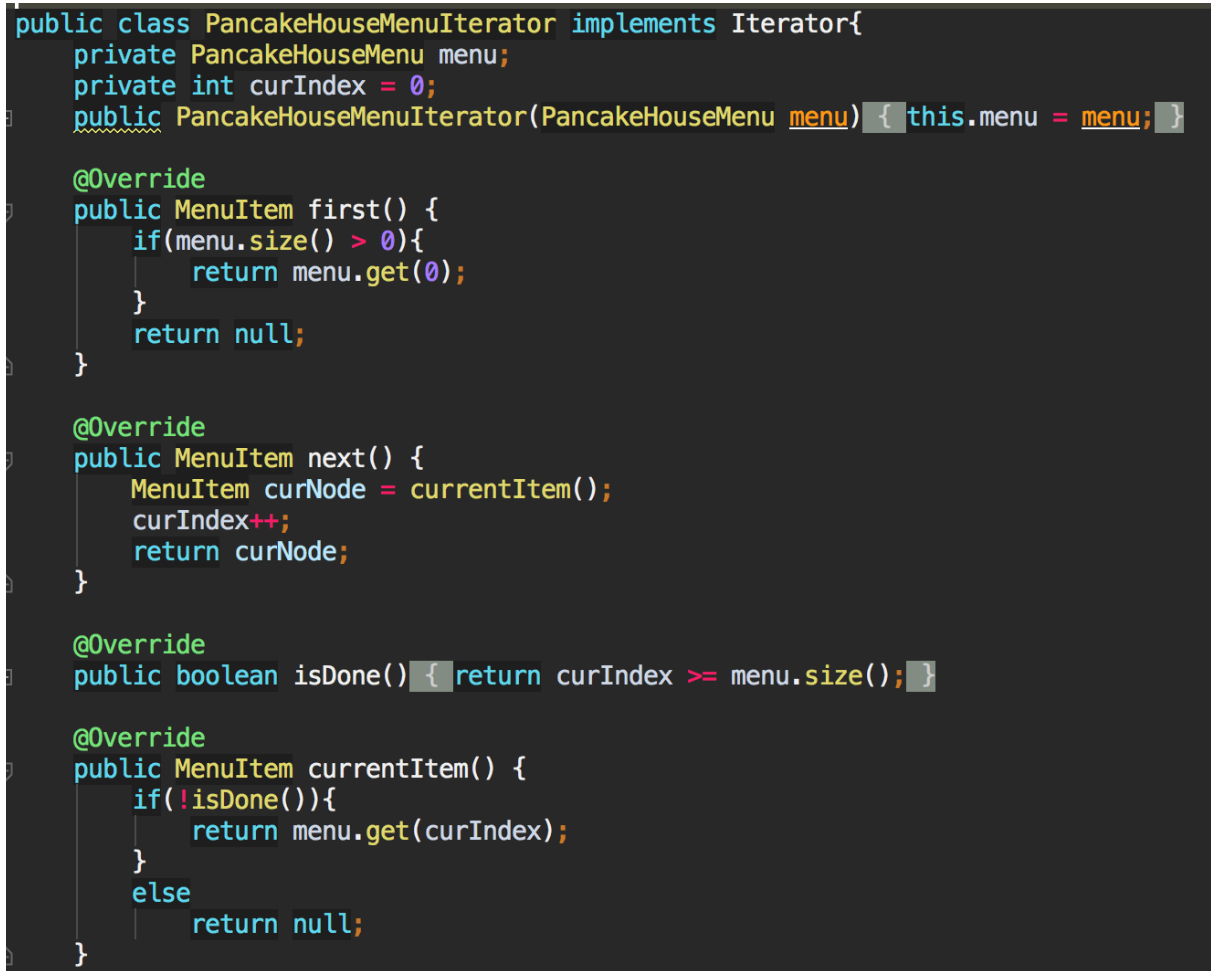
```
public interface Iterator {
    public MenuItem first();
    public MenuItem next();
    public boolean isDone();
    public MenuItem currentItem();
}
```

DinerMenuIterator



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

    public DinerMenuIterator(DinerMenu 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;
    }
```

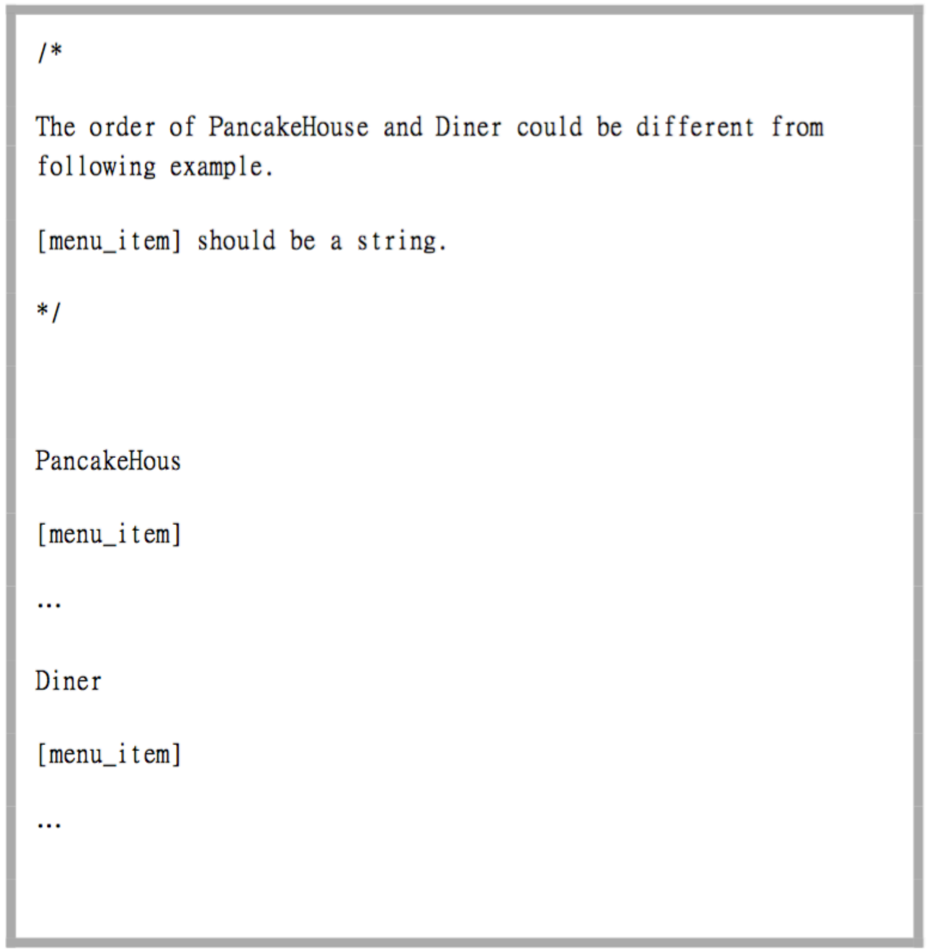
PancakeHouseMenuIterator



```
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;
    }
```

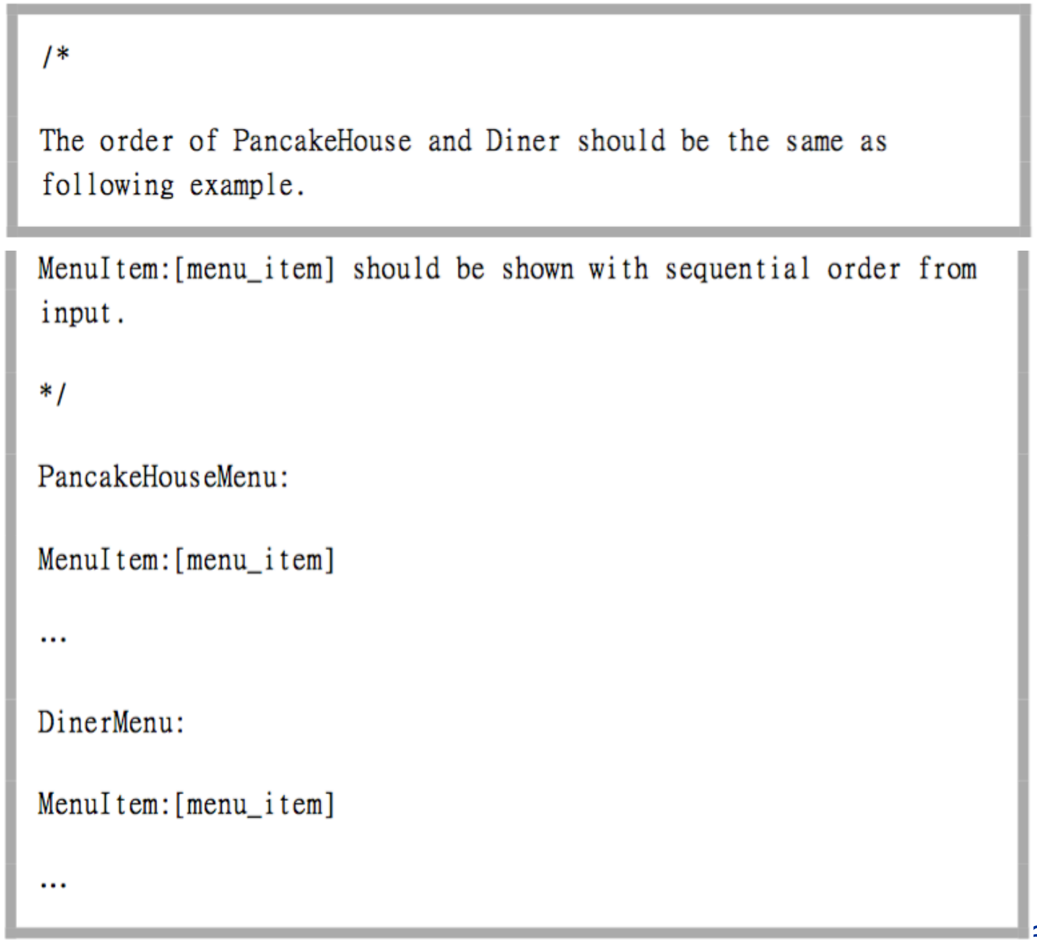
Input / Output

Input:



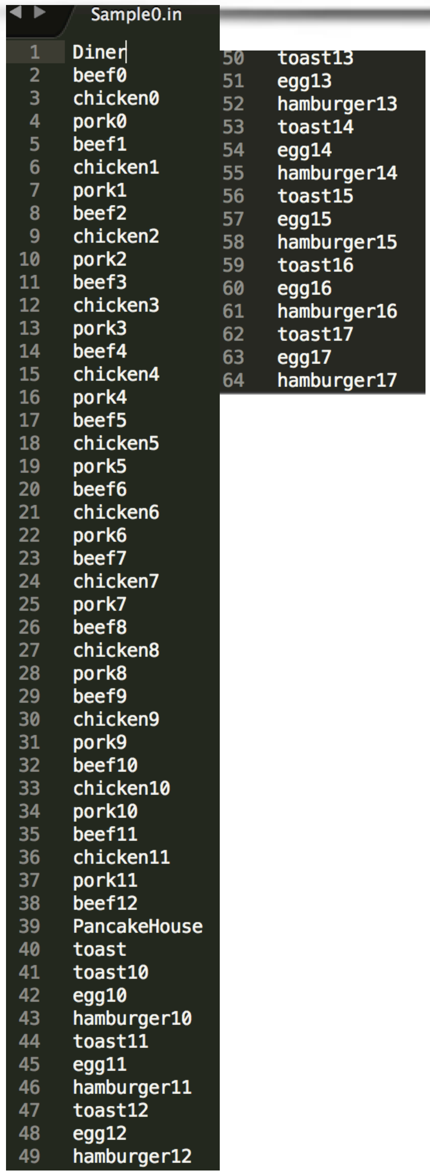
```
/*
The order of PancakeHouse and Diner could be different from
following example.
[menu_item] should be a string.
*/

PancakeHous
[menu_item]
...
Diner
[menu_item]
...
```

Output:



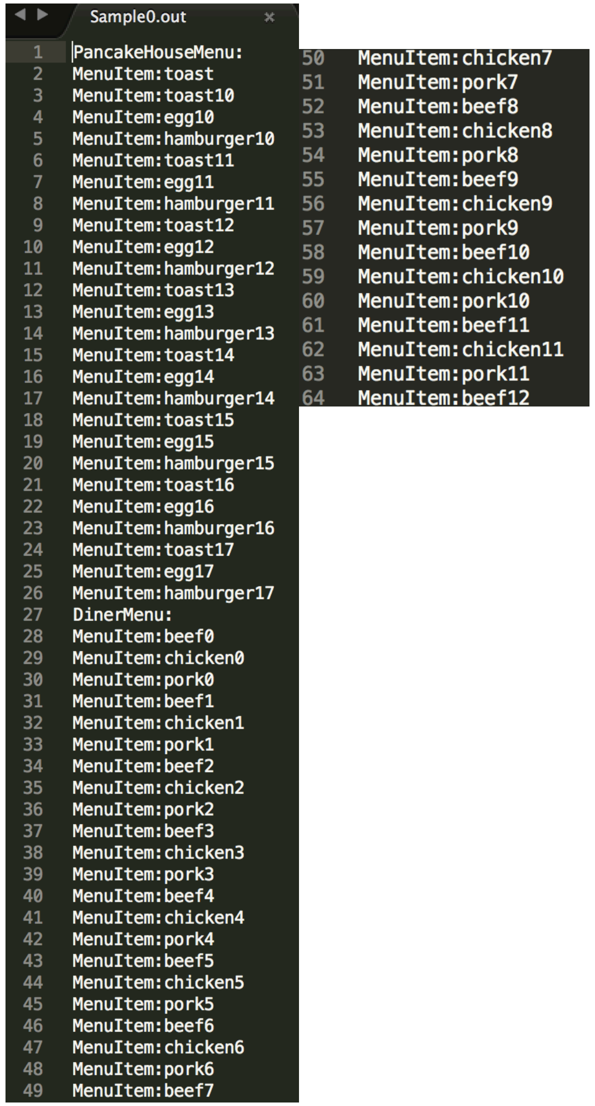
```
/*
The order of PancakeHouse and Diner should be the same as
following example.
MenuItem:[menu_item] should be shown with sequential order from
input.
*/
PancakeHouseMenu:
MenuItem:[menu_item]
...
DinerMenu:
MenuItem:[menu_item]
...
```

Test case



Sample0.in

```
Diner
beef0
chicken0
pork0
beef1
chicken1
pork1
beef2
chicken2
pork2
beef3
chicken3
pork3
beef4
chicken4
pork4
beef5
chicken5
pork5
beef6
chicken6
pork6
beef7
chicken7
pork7
beef8
chicken8
pork8
beef9
chicken9
pork9
beef10
chicken10
pork10
beef11
chicken11
pork11
beef12
PancakeHouse
toast
toast10
egg10
hamburger10
toast11
egg11
hamburger11
toast12
egg12
hamburger12
toast13
egg13
hamburger13
toast14
egg14
hamburger14
toast15
egg15
hamburger15
toast16
egg16
hamburger16
toast17
egg17
hamburger17
```



Sample0.out

```
PancakeHouseMenu:
MenuItem:toast
MenuItem:toast10
MenuItem:egg10
MenuItem:hamburger10
MenuItem:toast11
MenuItem:egg11
MenuItem:hamburger11
MenuItem:toast12
MenuItem:egg12
MenuItem:hamburger12
MenuItem:toast13
MenuItem:egg13
MenuItem:hamburger13
MenuItem:toast14
MenuItem:egg14
MenuItem:hamburger14
MenuItem:toast15
MenuItem:egg15
MenuItem:hamburger15
MenuItem:toast16
MenuItem:egg16
MenuItem:hamburger16
MenuItem:toast17
MenuItem:egg17
MenuItem:hamburger17
DinerMenu:
MenuItem:beef0
MenuItem:chicken0
MenuItem:pork0
MenuItem:beef1
MenuItem:chicken1
MenuItem:pork1
MenuItem:beef2
MenuItem:chicken2
MenuItem:pork2
MenuItem:beef3
MenuItem:chicken3
MenuItem:pork3
MenuItem:beef4
MenuItem:chicken4
MenuItem:pork4
MenuItem:beef5
MenuItem:chicken5
MenuItem:pork5
MenuItem:beef6
MenuItem:chicken6
MenuItem:pork6
MenuItem:beef7
MenuItem:chicken7
MenuItem:pork7
MenuItem:beef8
MenuItem:chicken8
MenuItem:pork8
MenuItem:beef9
MenuItem:chicken9
MenuItem:pork9
MenuItem:beef10
MenuItem:chicken10
MenuItem:pork10
MenuItem:beef11
MenuItem:chicken11
MenuItem:pork11
MenuItem:beef12
```