Iterator Pattern

Shin-Jie Lee (李信杰)
Associate Professor
Computer and Network Center
Department of CSIE
National Cheng Kung University

Design Aspect of Iterator

How an aggregate's elements are accessed, traversed

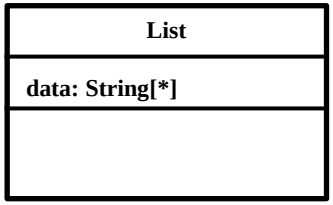
Outline

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

Print Out Items in Different Data Structures (Iterator)

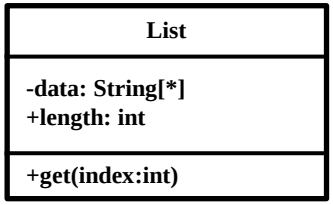
Requirements Statement$_1$

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



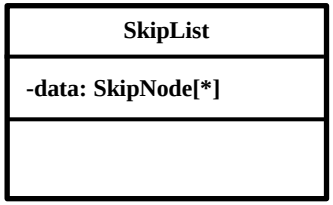
Requirements Statement$_2$

- 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.



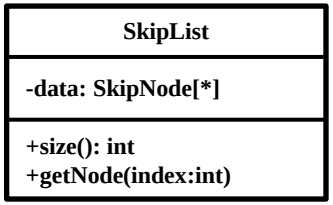
Requirements Statement$_3$

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



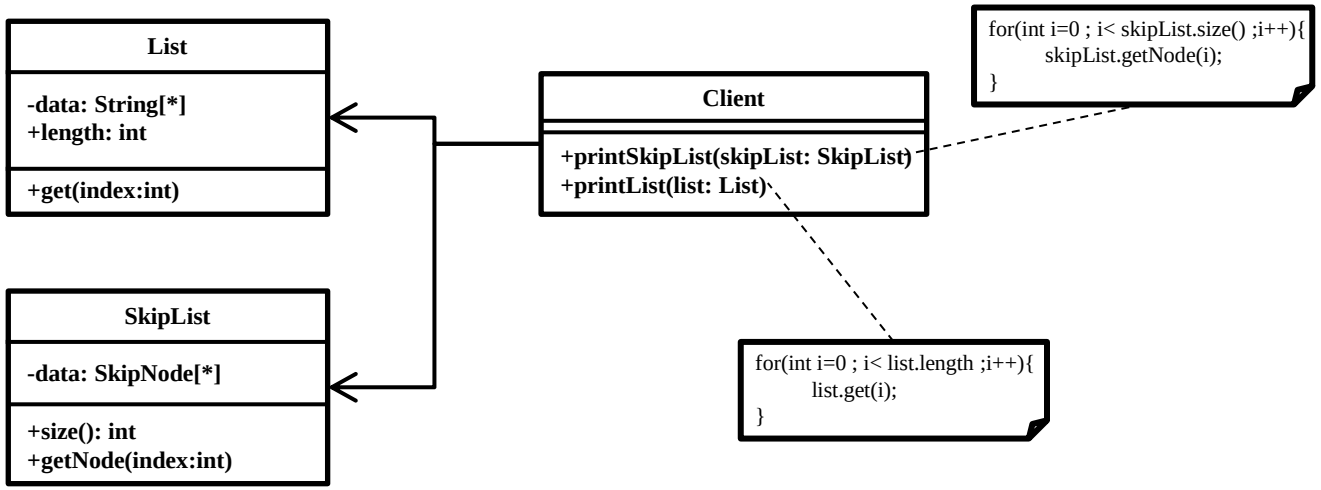
Requirements Statement$_4$

- 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.

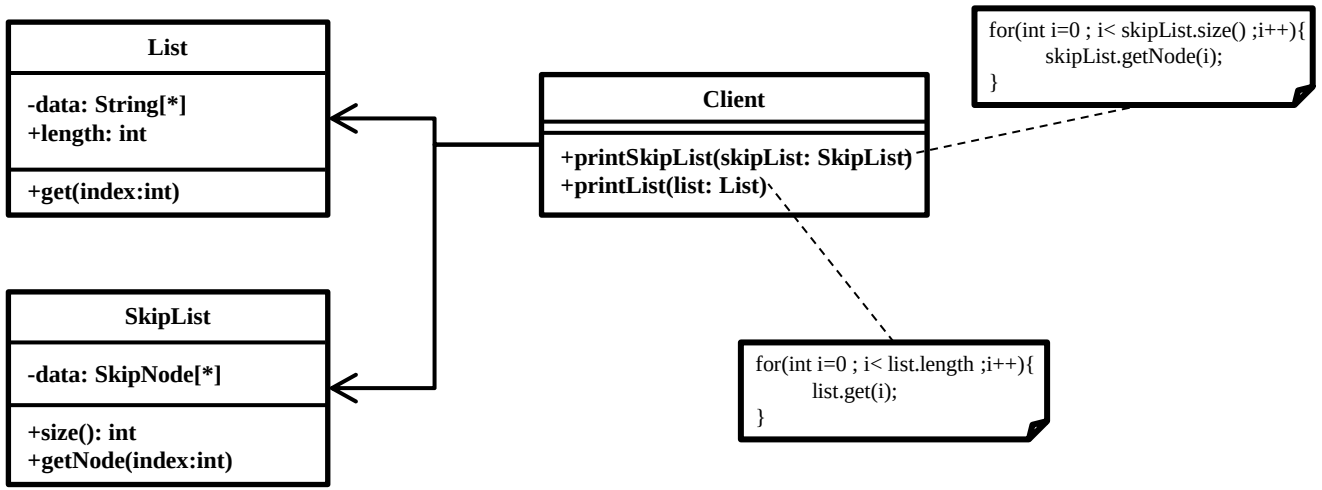


Requirements Statement$_5$

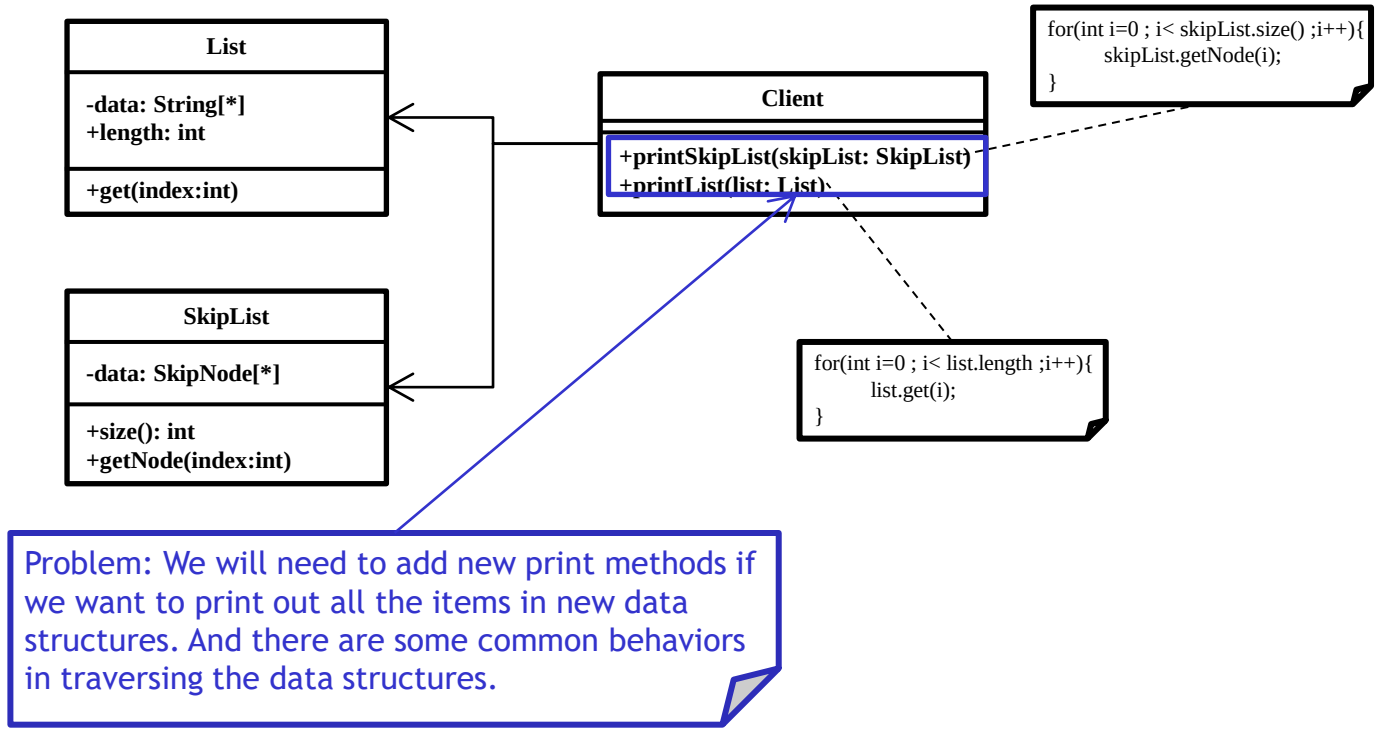
- Now we have to traverse both List and SkipList to print out those object items in the two different data structures for some purpose.



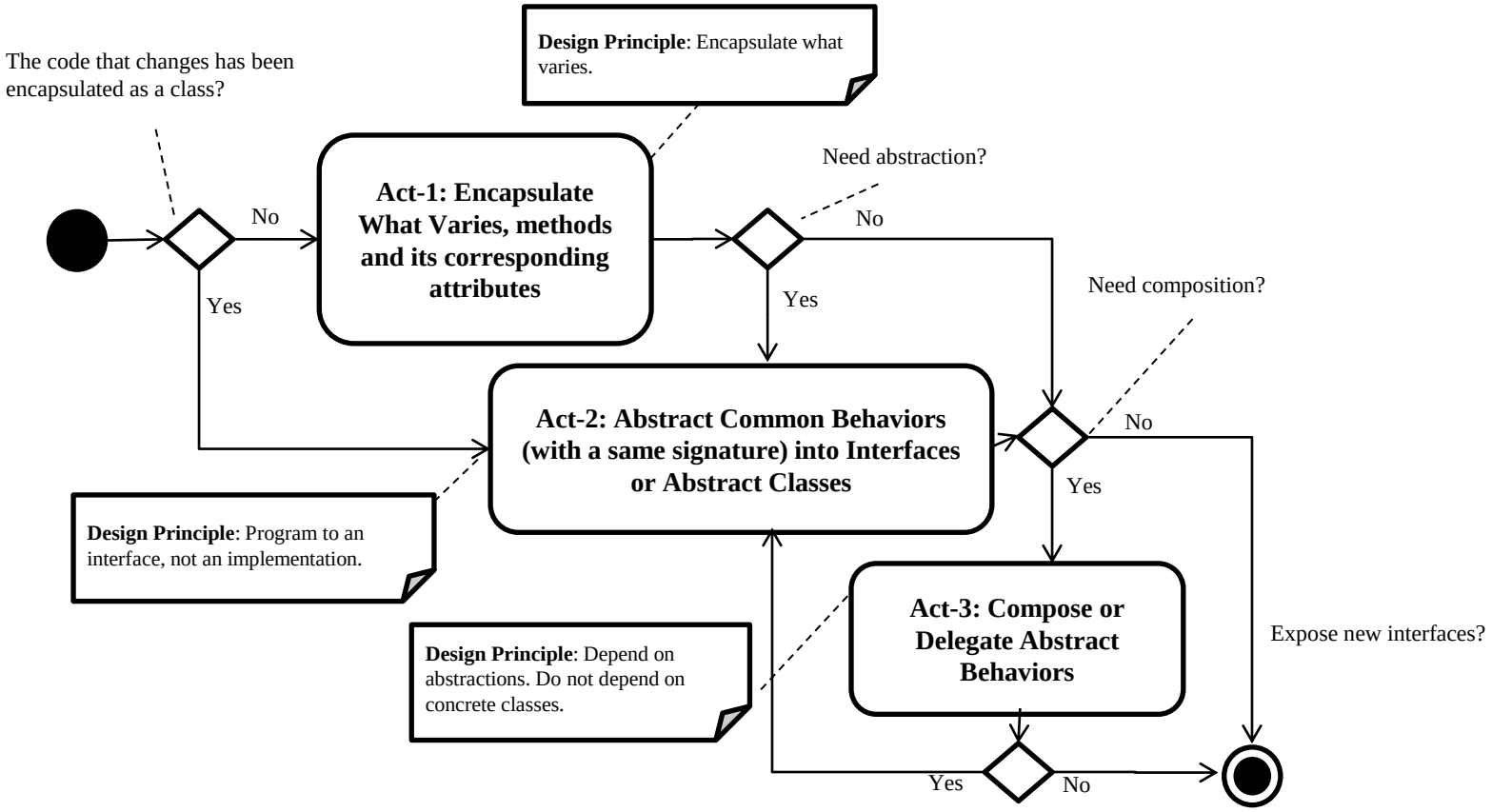
Initial Design



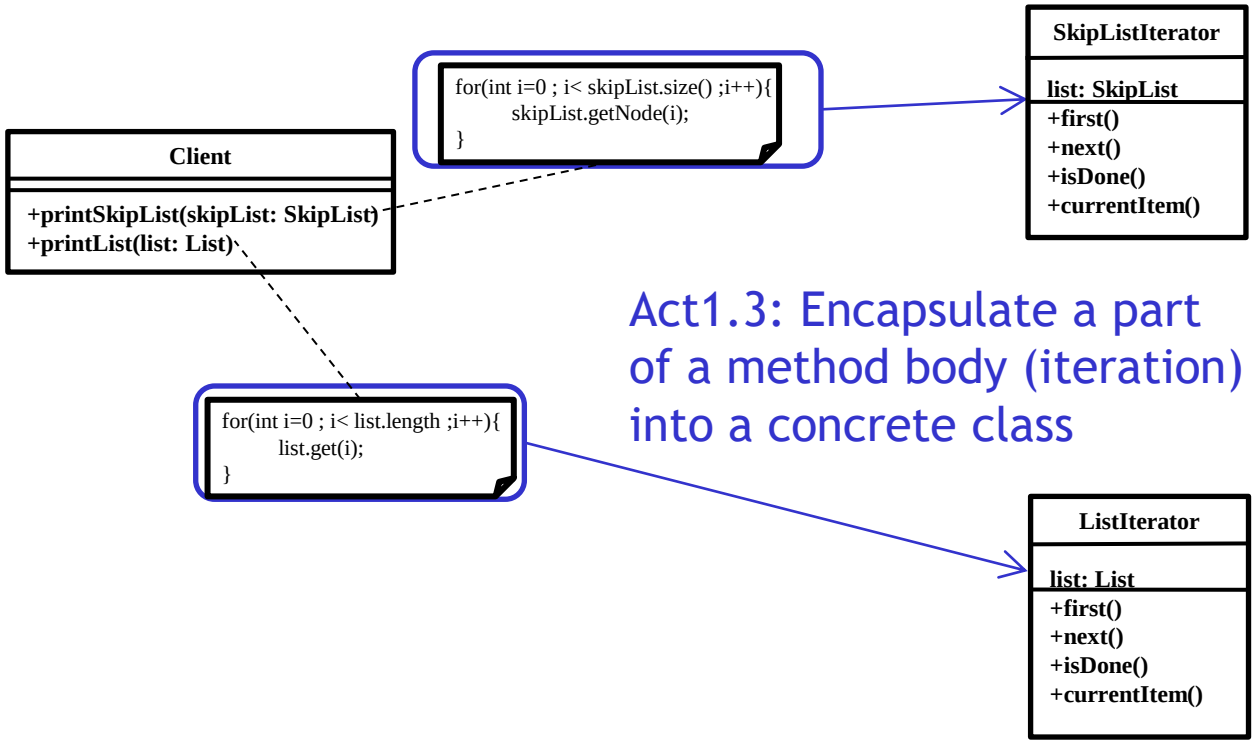
Problems with Initial Design



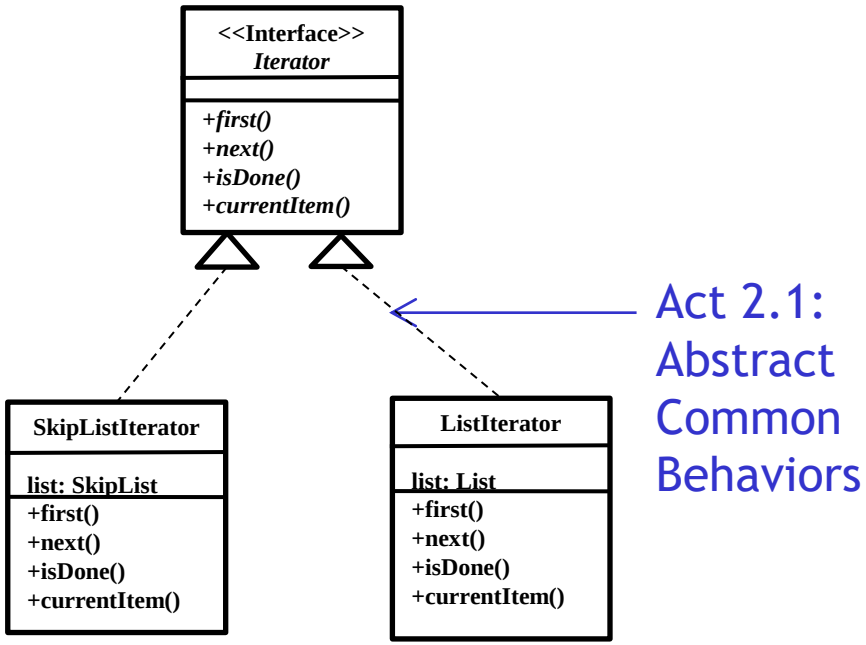
Design Process for Change



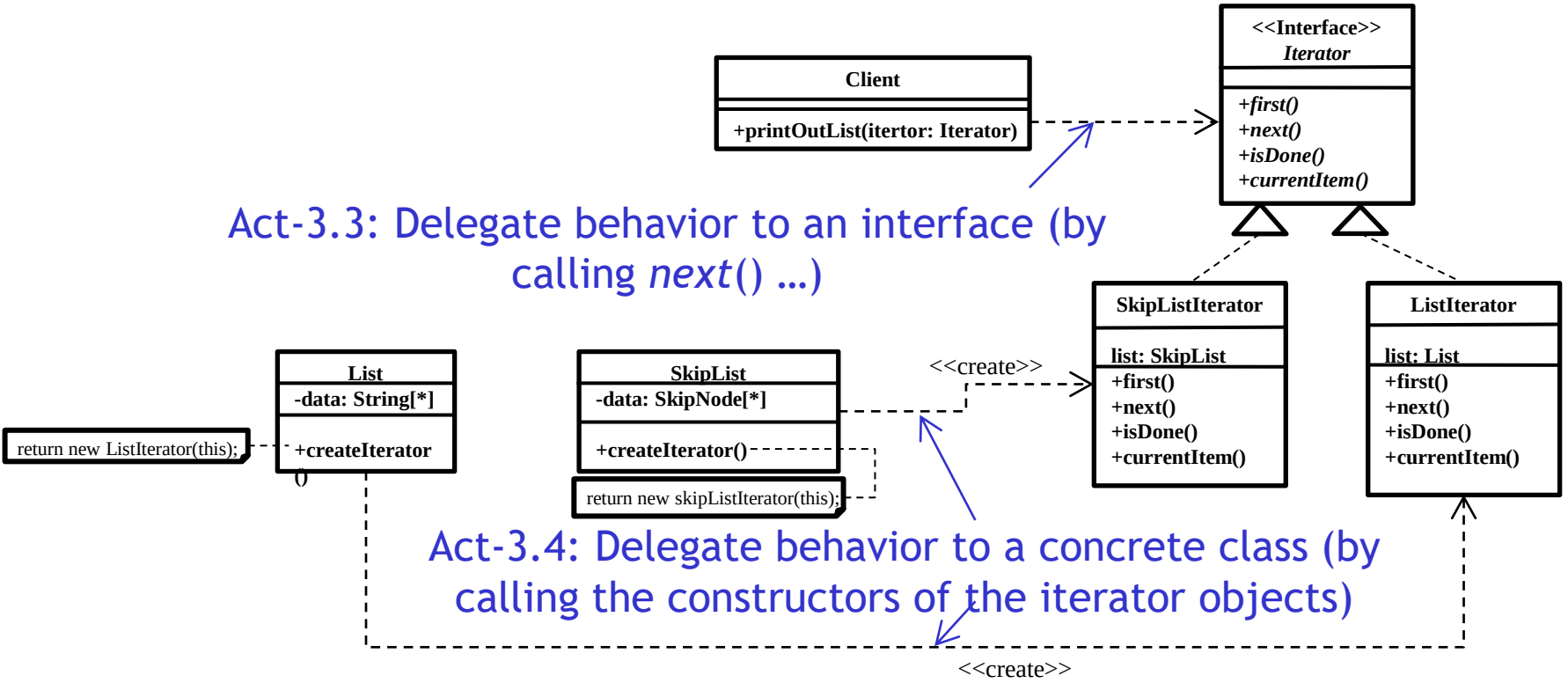
Act-1: Encapsulate What Varies



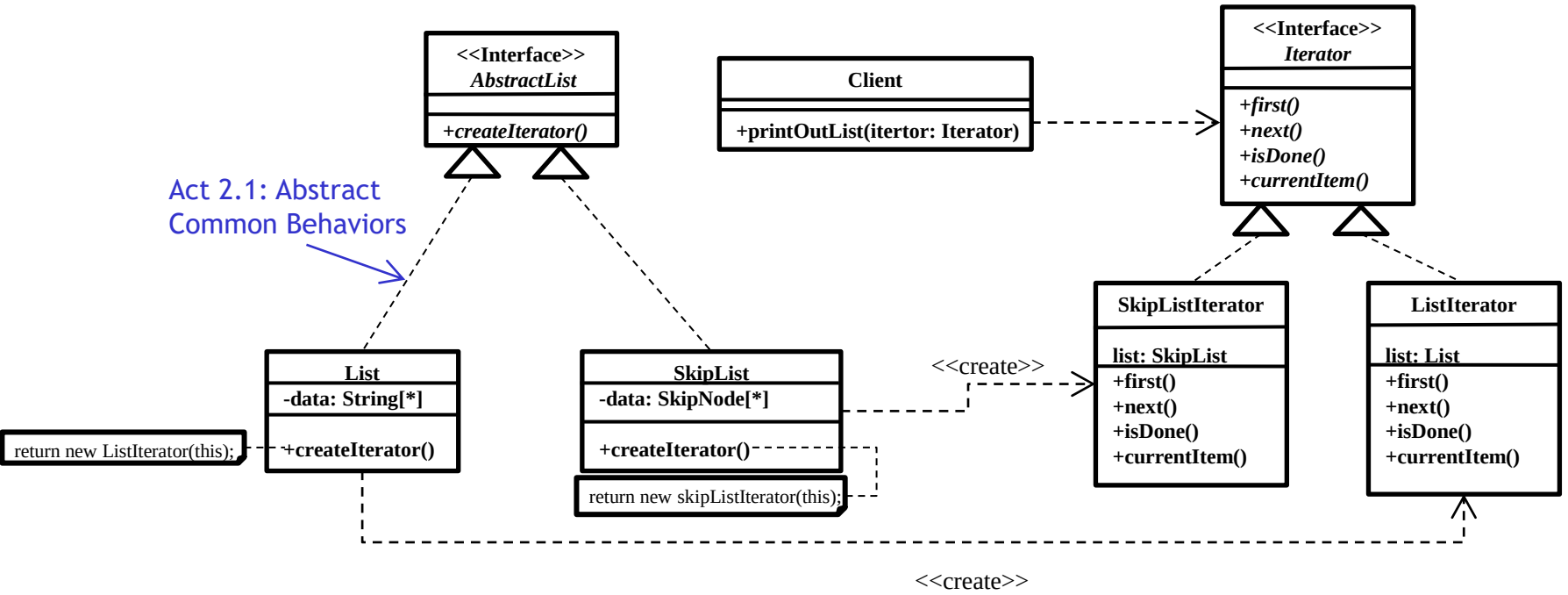
Act-2: Abstract Common Behaviors



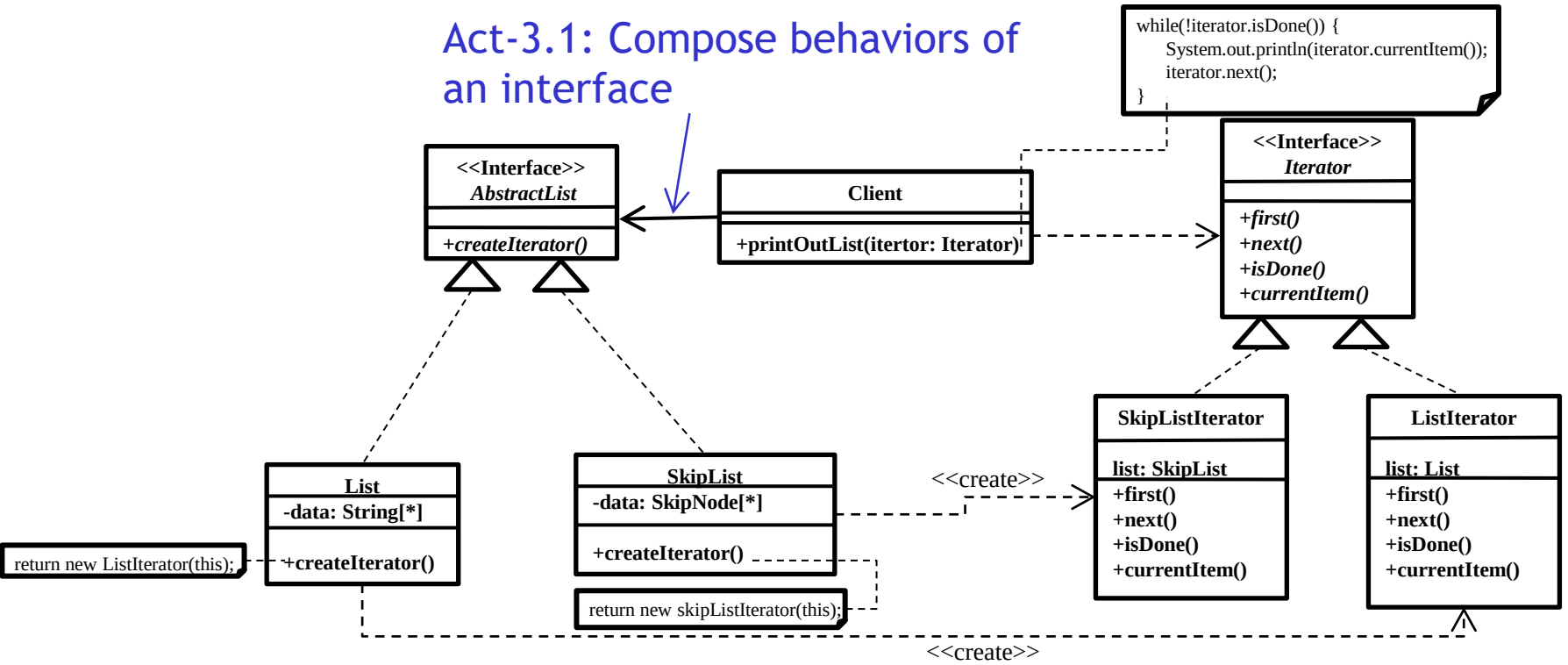
Act-3: Compose Abstract Interfaces/Abstract Classes or Delegate Behaviors



Act-2: Abstract Common Behaviors



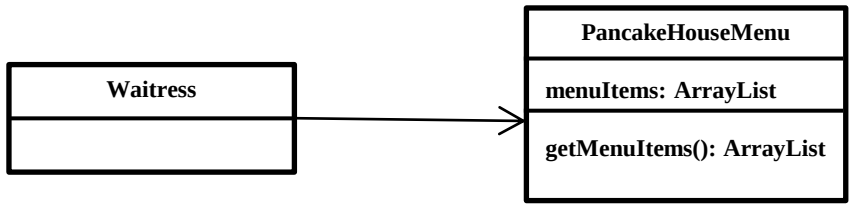
Act-3: Compose Abstract Interfaces/Abstract Classes or Delegate Behaviors



Merge Two Menus

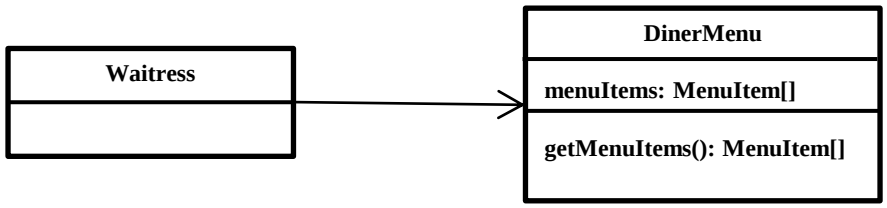
Requirements Statement

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



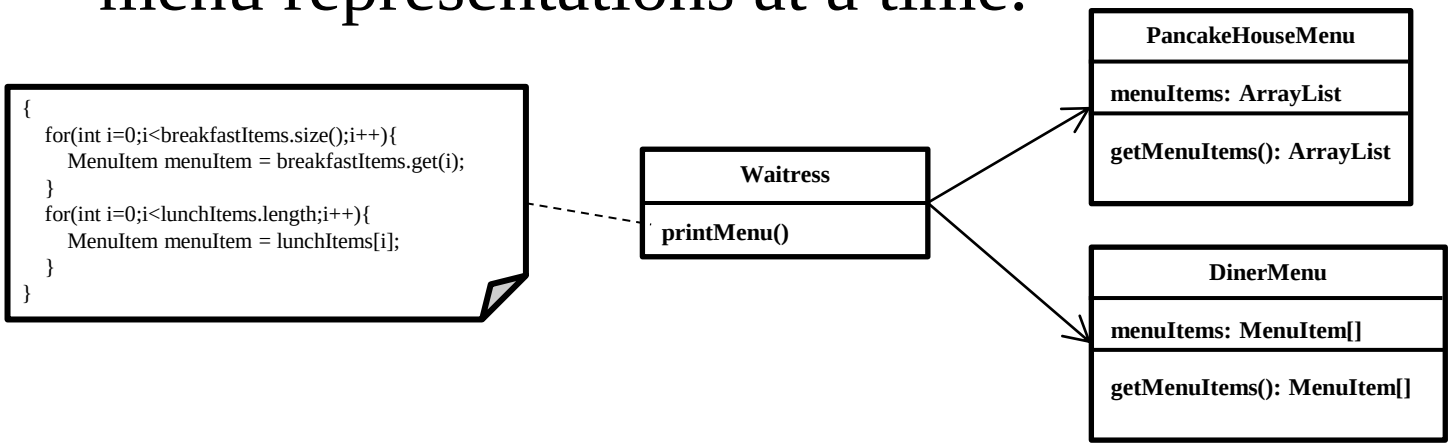
Requirements Statement

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

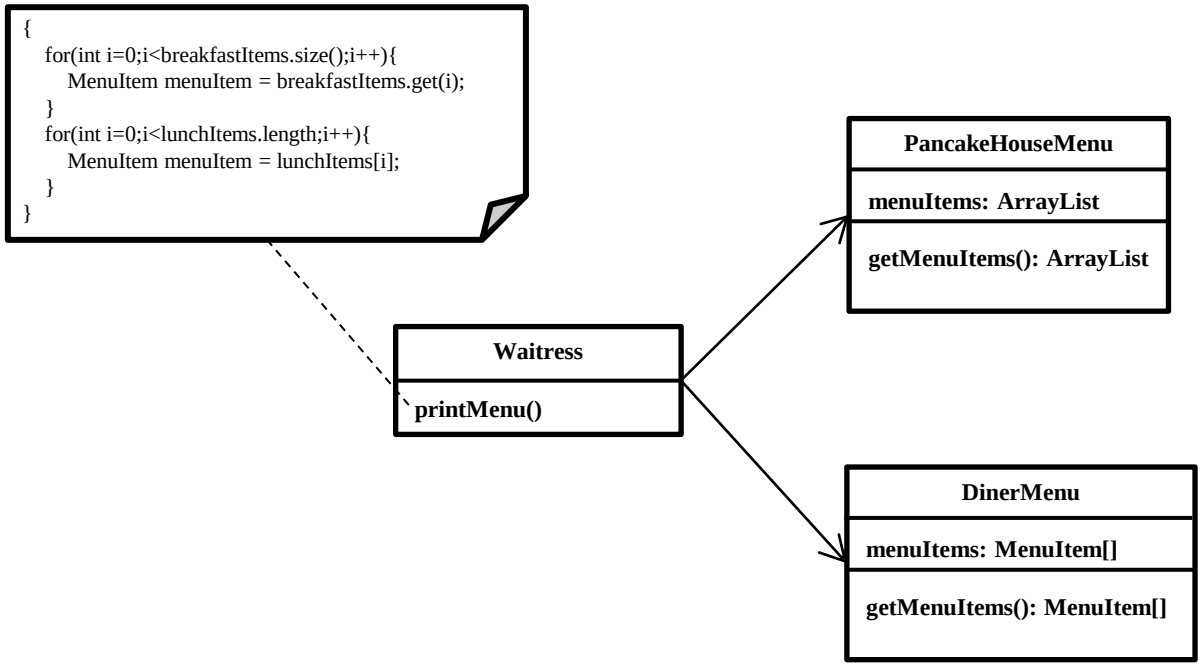


Requirements Statement

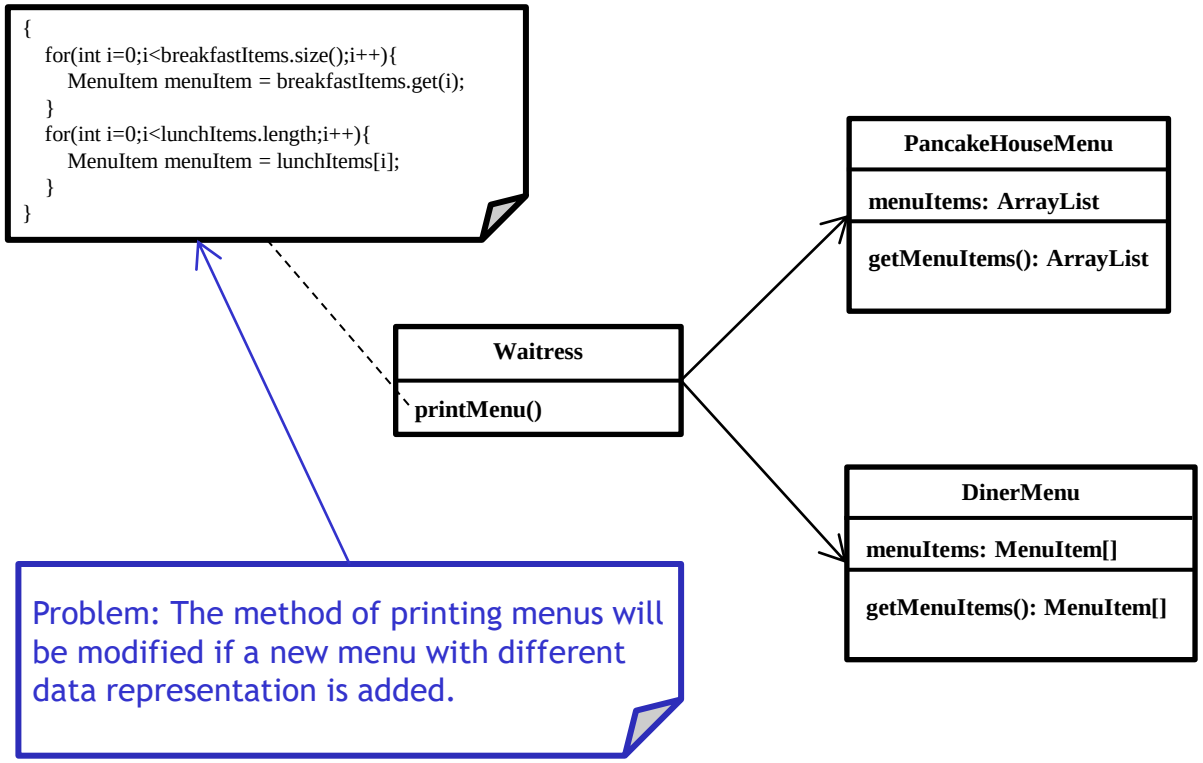
- Now, these two restaurants are merged and intend to provide service in one place, so a waitress should keep both menus in hands.
- The waitress would like to print two different menu representations at a time.



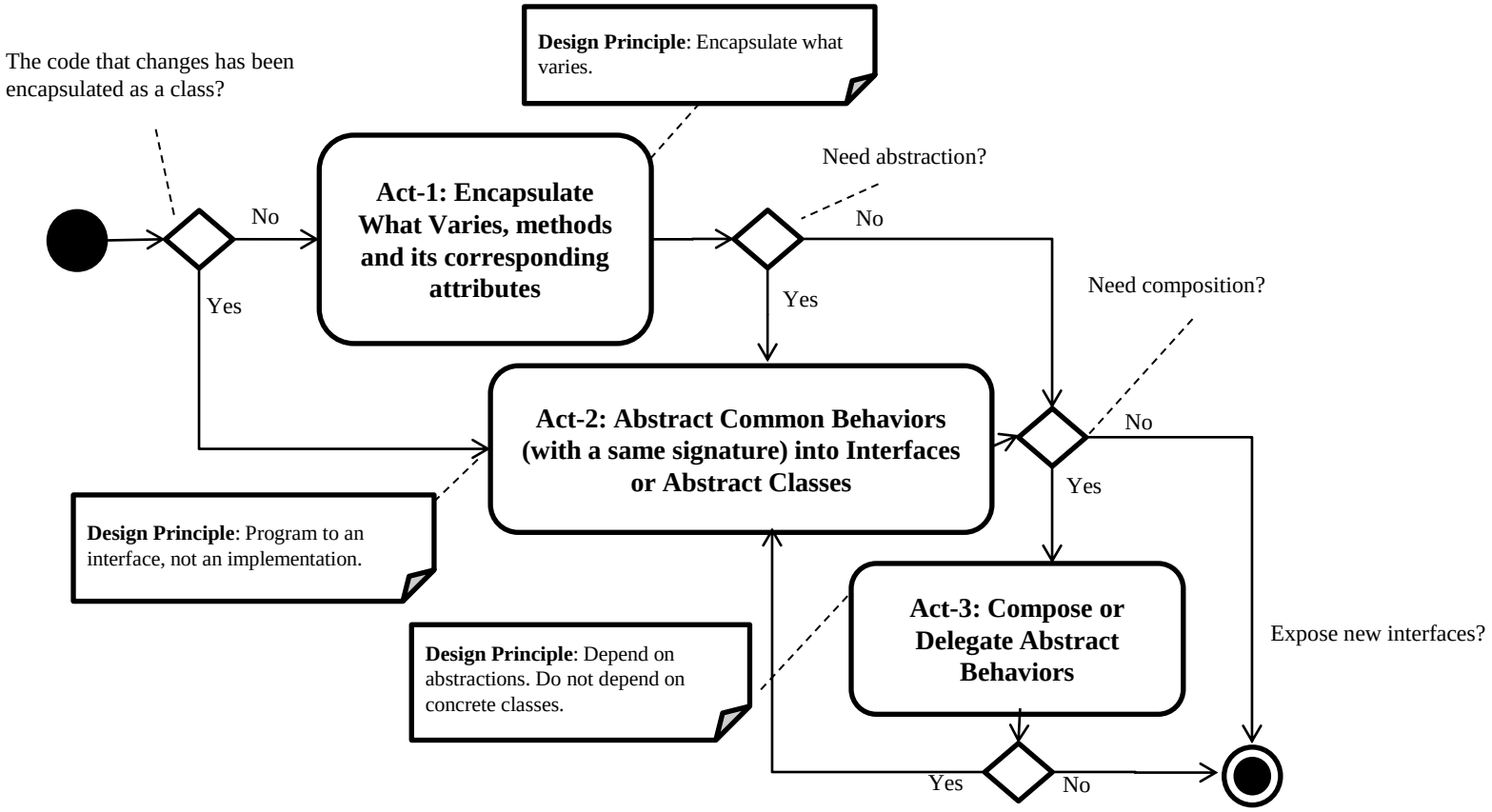
Initial Design - Class Diagram



Problems with Initial Design



Design Process for Change



Act-1.3: Encapsulate What Varies

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

```
{
    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];
    }
}
```

Waitress

printMenu()

PancakeHouseMenuIterator

- menu: PancakeHouseMenu

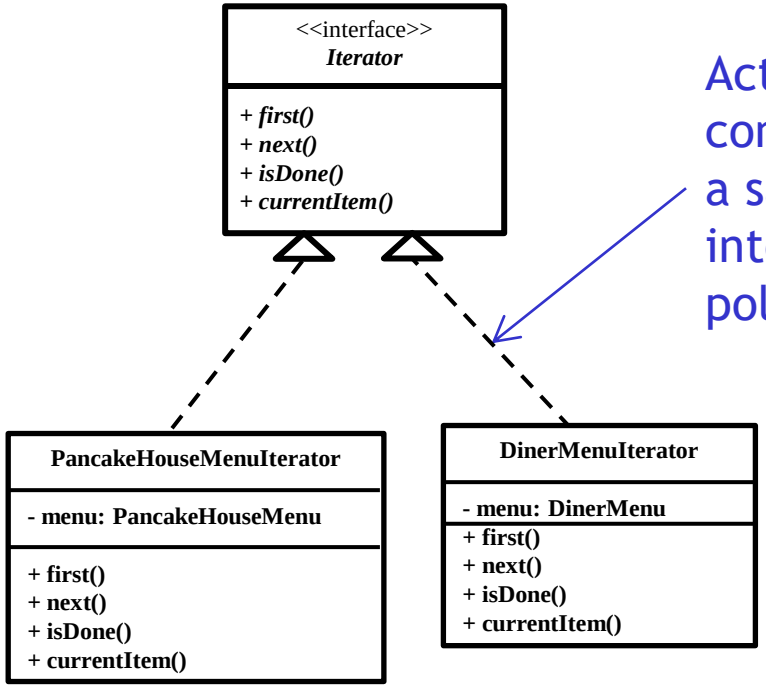
+ first()
+ next()
+ isDone()
+ currentItem()

DinerMenuIterator

- menu: DinerMenu

+ first()
+ next()
+ isDone()
+ currentItem()

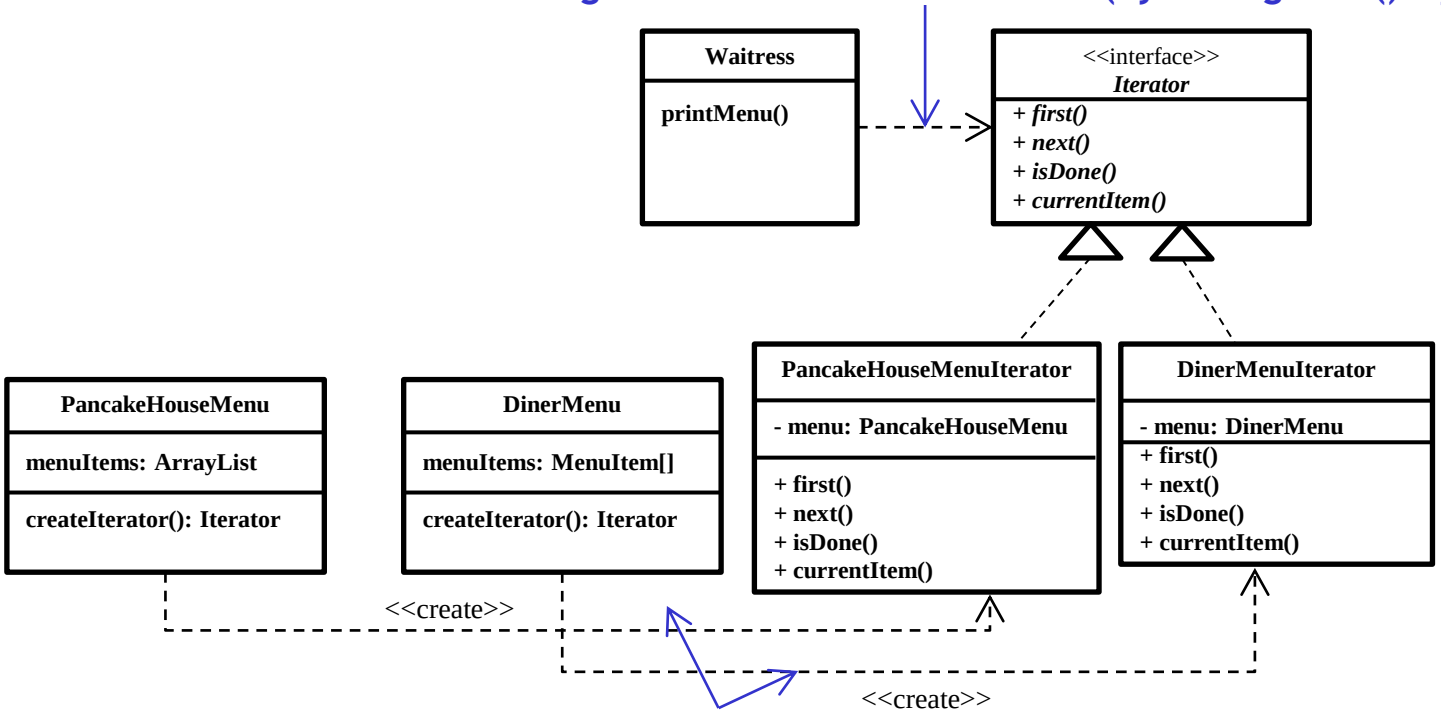
Act-2.1: Abstract Common Behaviors



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

Act-3: Delegate Behaviors

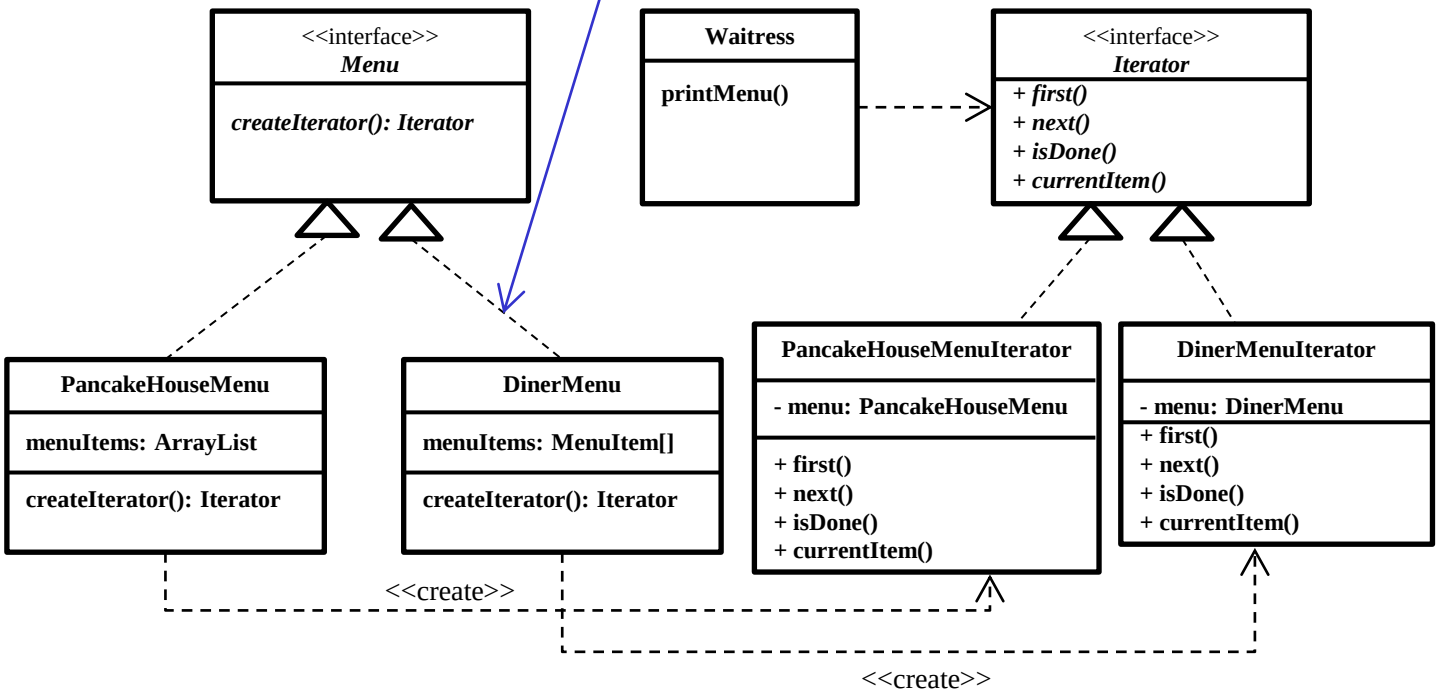
Act-3.3: Delegate behavior to an interface (by calling *next*() …)



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

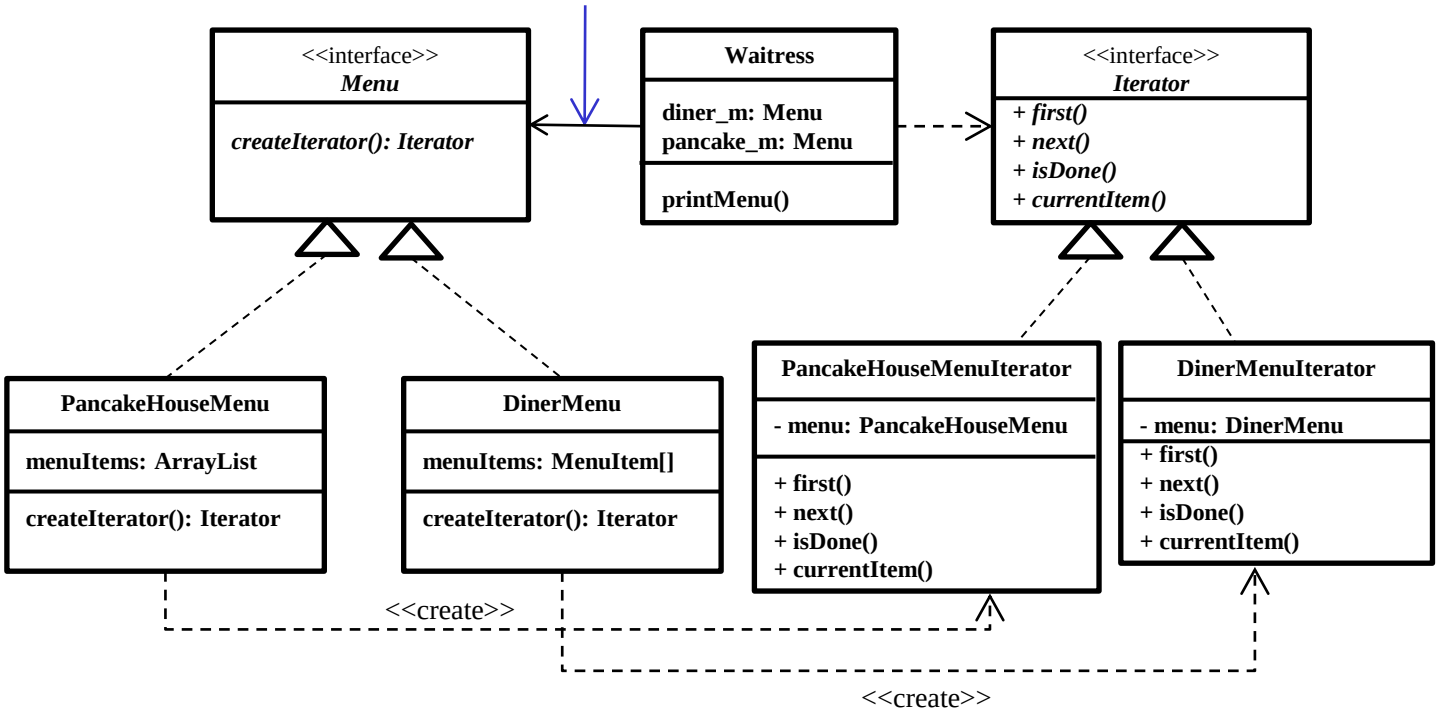
Act-2: Abstract Common Behaviors

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

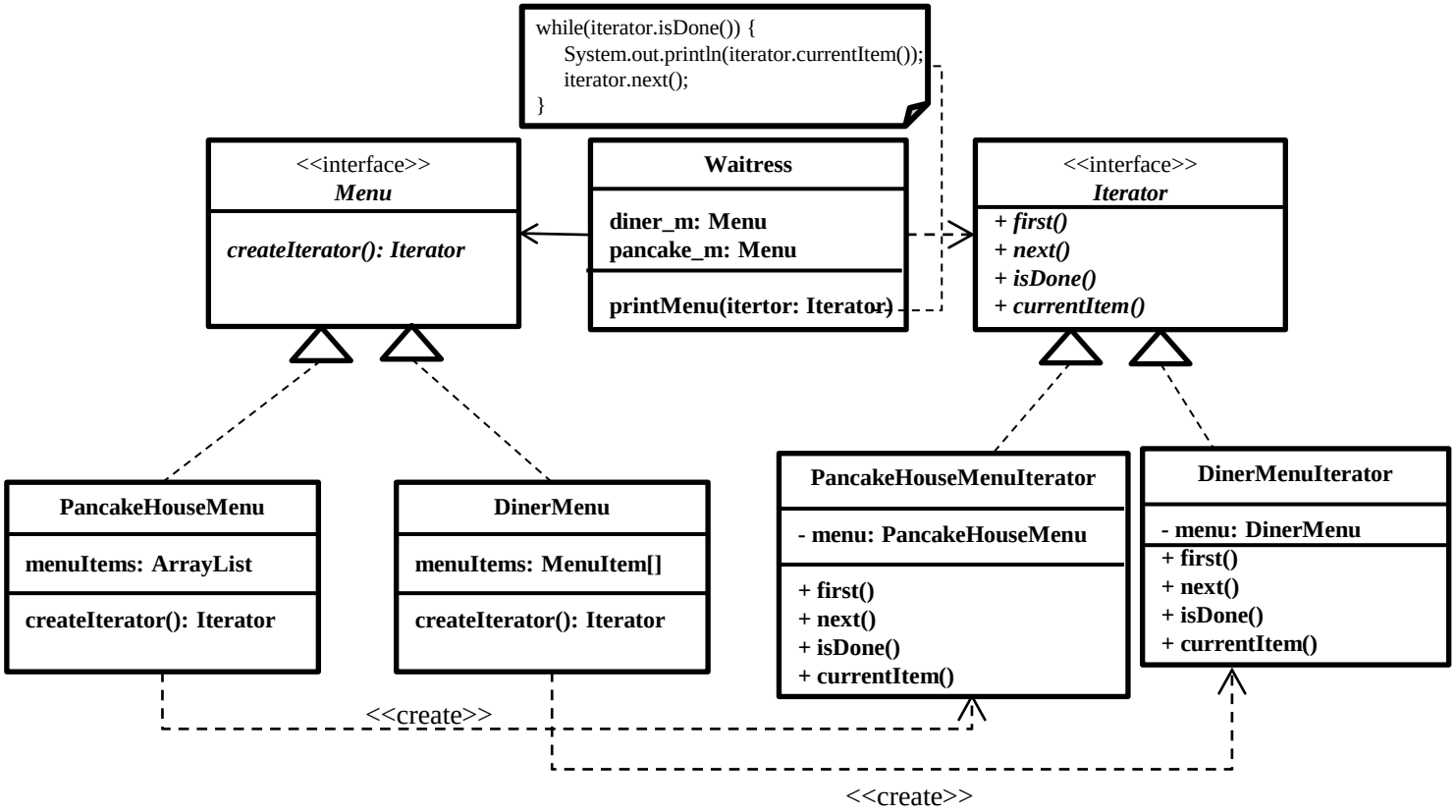


Act-3: Compose Abstract Interfaces/Abstract Classes

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



Refactored Design after Design Process



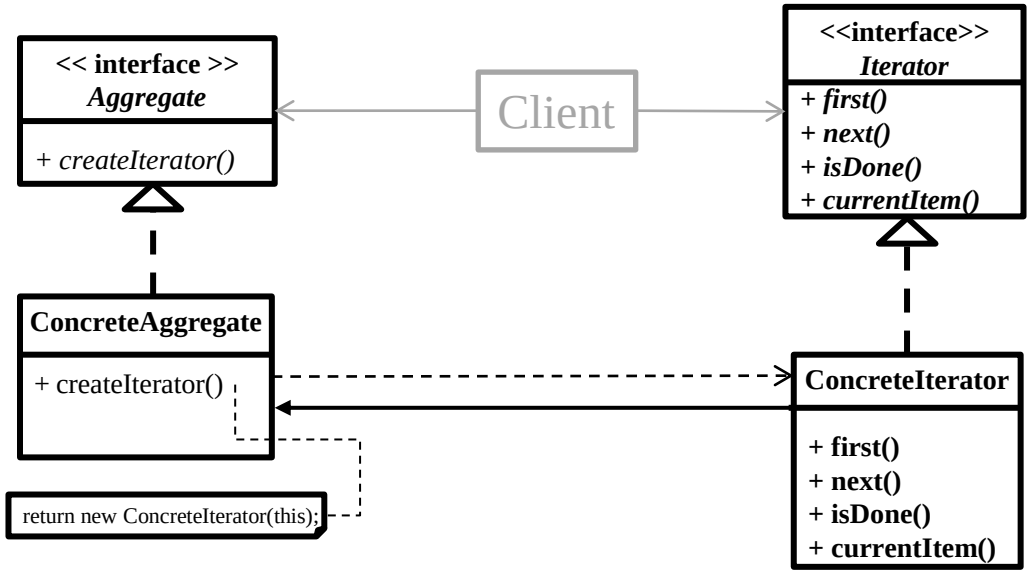
Recurrent Problem

- 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.

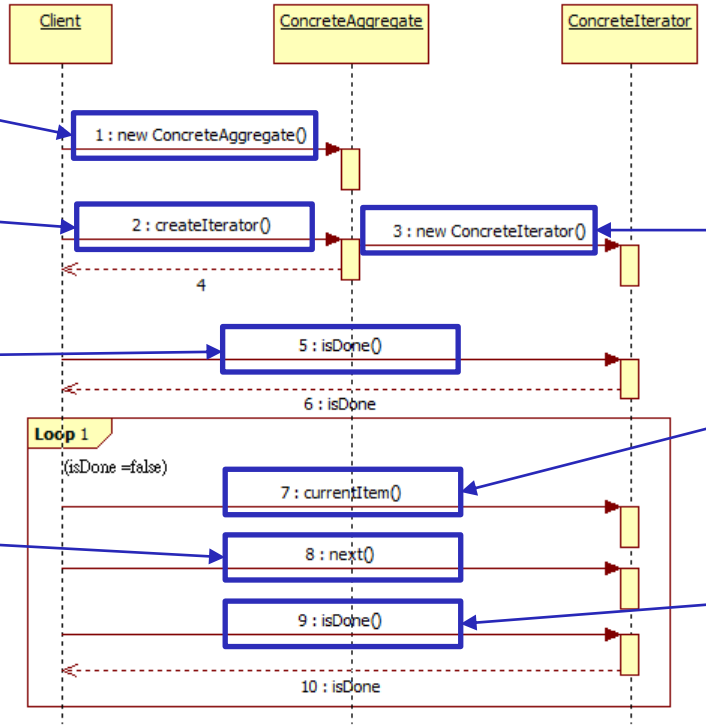
3. ConcreteAggregate creates a ConcreteIterator.

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.