Iterator Pattern

Lect 25 --26

30-10-2023

Background

- Traversing through elements in an aggregation is a very common problem in programming:
  - **Iterators provide a uniform way of doing this …**
- Using an iterator, we don't need to know:
  - How the collection is implemented!
  - Concurrent access and Synchronization
  - Different methods for iterating

Iterator Pattern: Intent

- Provides a way to access the elements of an aggregate object sequentially:
  - Without exposing the programmer to the complexity of underlying representation.
- Moves the responsibility for access and traversal:
  - **From the aggregate object to an iterator object.**

Iterator Pattern: Pros

- Supports the same interface for all aggregates.
- Supports multiple types of traversals of aggregate objects.
- Supports multiple iterators…

Iterator: Essential Idea

- The elements of a collection:
  - **Accessed in some sequential order that may be independent of their position in the collection.**
- For example:
  - Level order: Access each object of a tree from left to right at each tree level
  - Inorder, post order, preorder, etc

The Iterator Pattern: Context

- It might be necessary to have more than one type of traversal on the same aggregate object.
  - **Also, not all types of traversals can be anticipated a priori.**

- **One should not bloat the interface of an aggregate object with all possible traversals.**

Iterator: Basic Methods

- **Reset:**
  - Move to the beginning of the range of elements
- **next:**
  - Advance to the next element
- **Get:**
  - Return the value referred to
- **hasNext:**
  - Interrogate it to see if it is at the end of the range

Iterator: Two Most Common Methods

- **public boolean hasNext()**
  - Returns true if the Iterator object has more elements (objects) that have not yet been returned by next()
- **public Object next()**
  - Returns the next element (object) from the iteration and advances to next element.

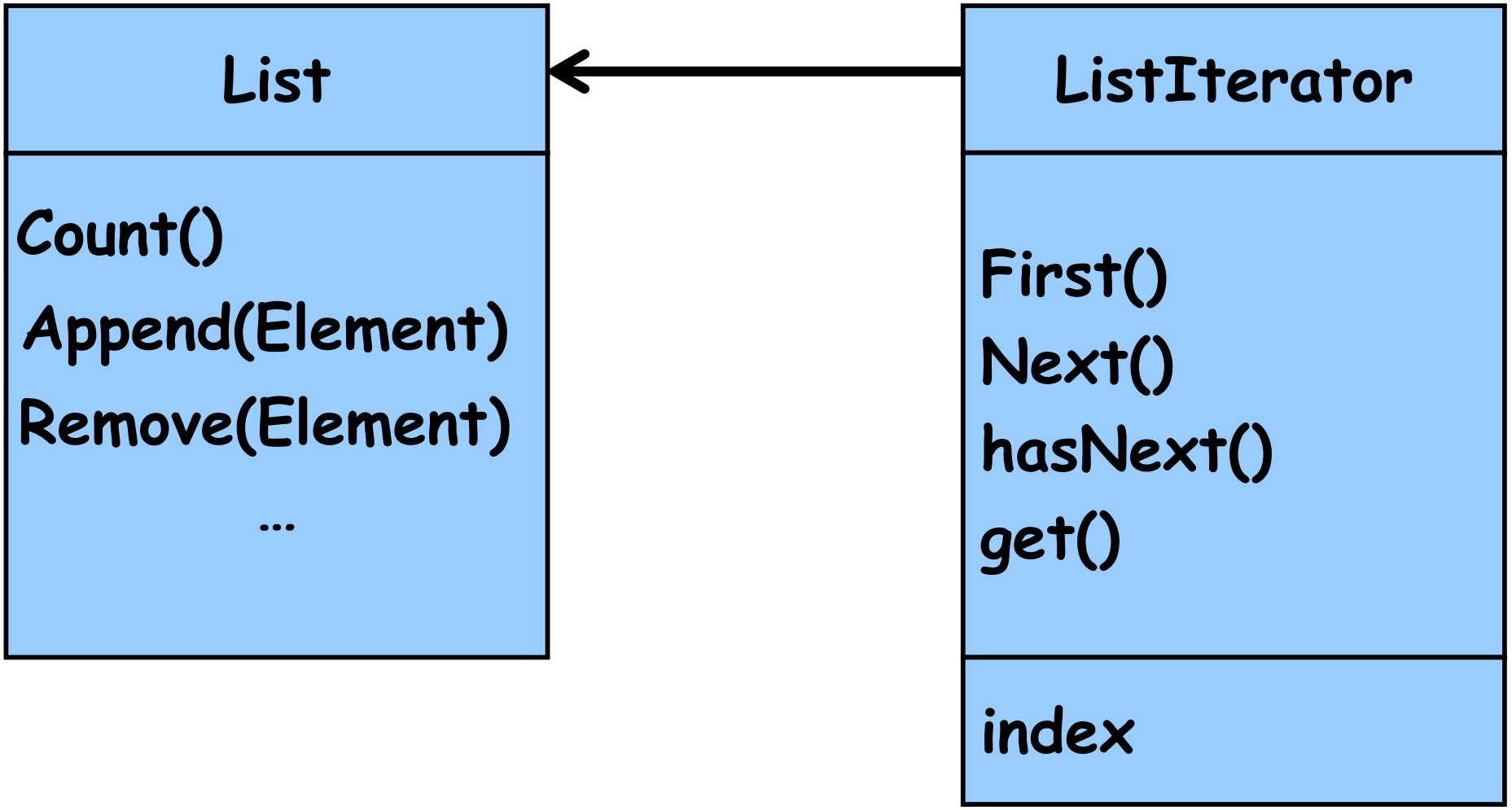
Additional Methods

- **Remove:** Current element is removed, as in Java List.
- **Add:** Add an element at the current position.
- **Count:** Return the number of elements in the aggregate.
- **First:** Reset

next() in Java: Issues

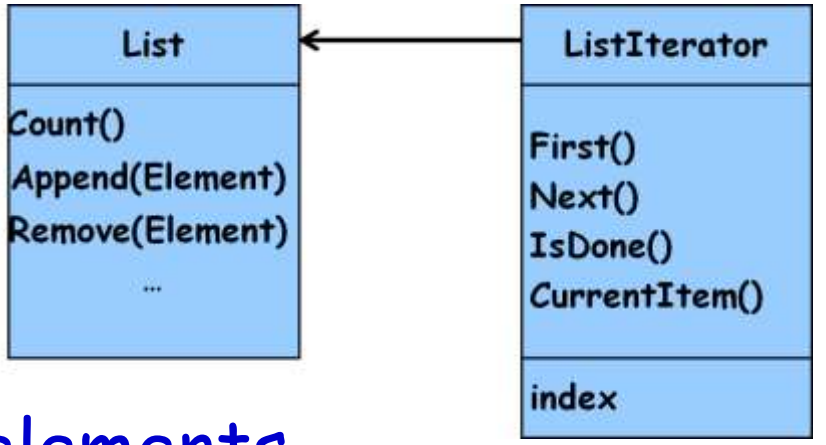
- Theoretically, when an iterator returns an element of the collection:
  - **It might return a clone of the element, or it might return a reference to the element**
- Iterators for Java collection classes **ArrayList<T> and HashSet<T>**:
  - **Actually return references**
- **Therefore, modifying the returned value will modify the element in the collection**

Iterator Pattern: Main Idea



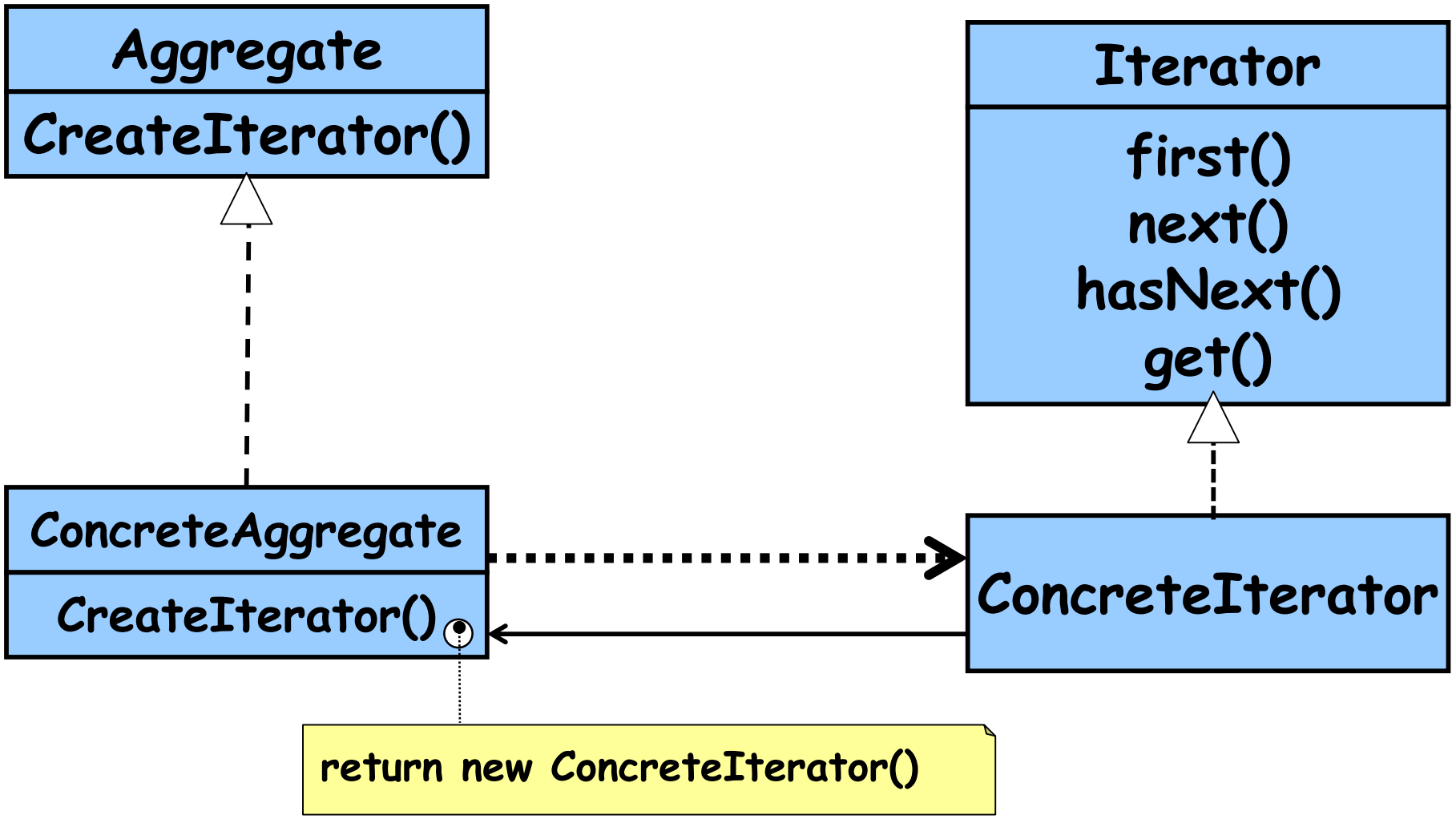
Iterator Pattern: Solution

- The responsibility for accessing and traversing an aggregation object:
  - **Placed in a separate iterator object and not in the aggregate object.**



- An iterator object should:
  - Provide the same interface for accessing and traversing elements,
  - Regardless of the class of the aggregate object and the kind of traversal performed

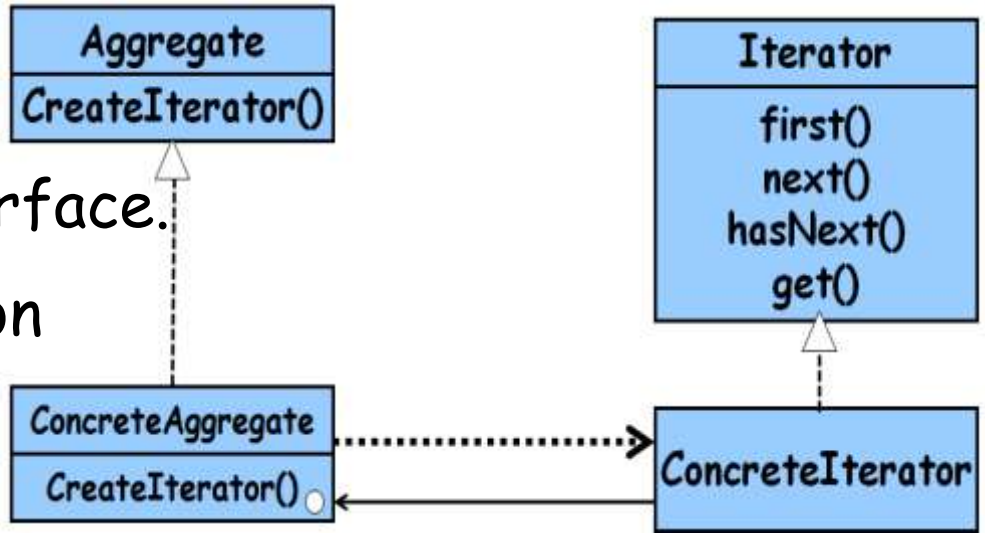
Structure of Iterator Pattern



A **ConcreteIterator** object holds a reference to the **ConcreteAggregate** object that created it.

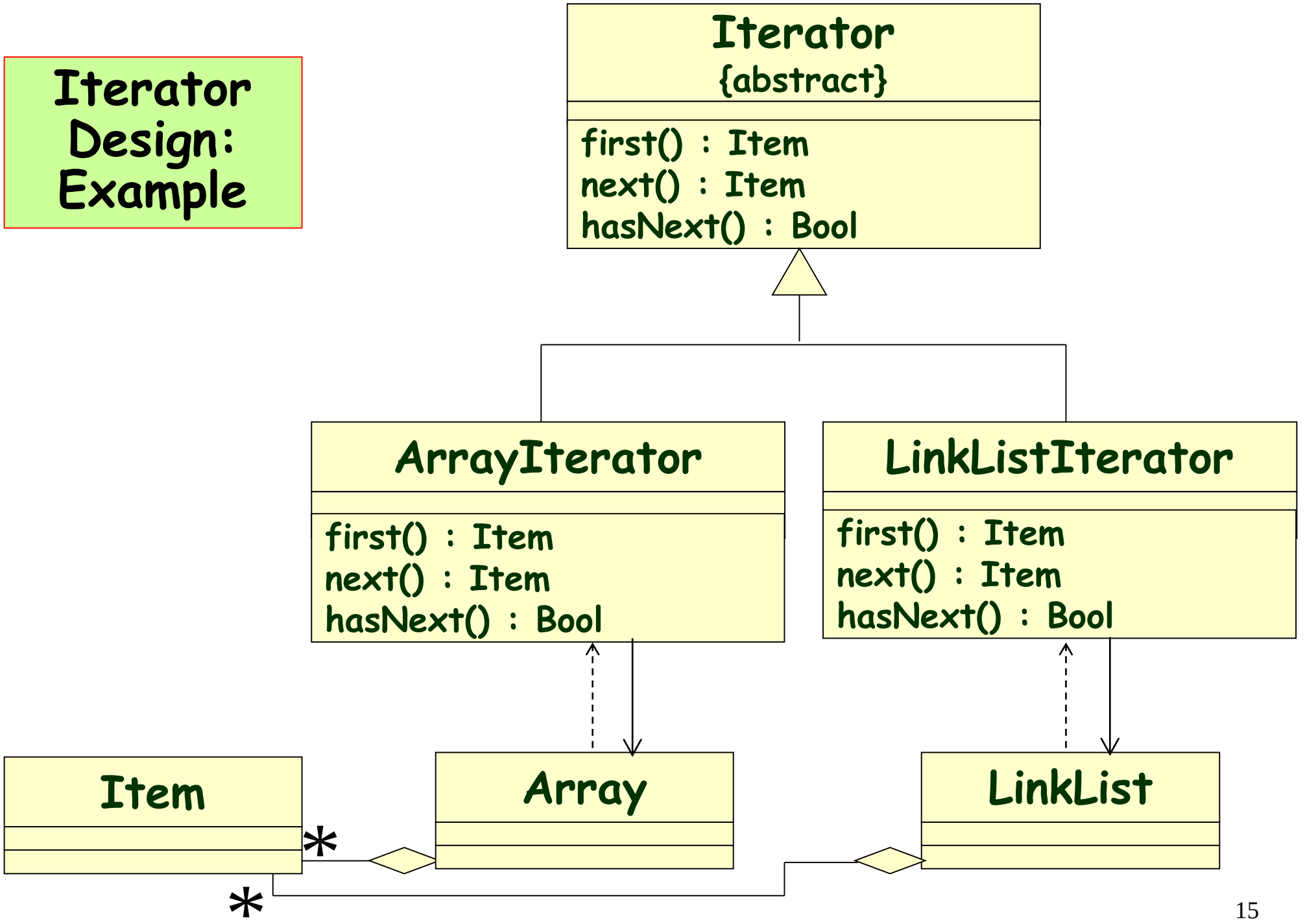
Iterator: Participants

- **Iterator**
  - Defines an interface for accessing and traversing elements
- **ConcreteIterator**
  - Implements the Iterator interface.
  - Keeps track of current position in the aggregate.



- **Aggregate**
  - Defines an interface for creating an Iterator object
- **ConcreteAggregate**
  - Creates and returns an instance of the ConcreteIterator

Iterator Design: Example



Iterator: Example

```
public class ArrayIterator
  implements Iterator {

  private String[] data;
  private int index;

  public ArrayIterator
  (String[] data) {

      this.data = data;

      this.index = 0;

  }
```

```
  public String first() {
      index = 0;
       return data[0];
  }
  public String next() {
      index++;
      return data[index];
  }
  public boolean hasNext() {
    if (index >= data.length)
          return false;
    return true;
  }
}
```

Iterator Design: Example

```
public class
LinkListIterator
implements Iterator {

private LinkListElement
first;

private LinkListElement
current;

public LinkListIterator
(LinkListElement first){

    this.first = first;

    this.current = first;

}
```

```
public String first() {
return first.value();
}
public String next() {
current = current.getNext();
return current.value();
}
public boolean moreElements()
{
if (current.getNext() == null)
    return false;
return true;
}
}
```

Iterating Through A Vector: without Iterator

```
import java.util.*;

public class IterationExample1{

  public static void main(String args[])  {

      Vector v = new Vector();

      // Put some Complex number objects in the vector.

      v.addElement(new Complex(3.2, 1.7));
```

Continued…

Iterating Through A Vector

```
        v.addElement(new Complex(7.6));
        v.addElement(new Complex());
        v.addElement(new Complex(-4.9, 8.3));
        for (int i = 0; i < v.size(); i++) {
            Complex c = (Complex)v.elementAt(i);
            System.out.println(c);}
    }
}
```

Iterating Through A Vector

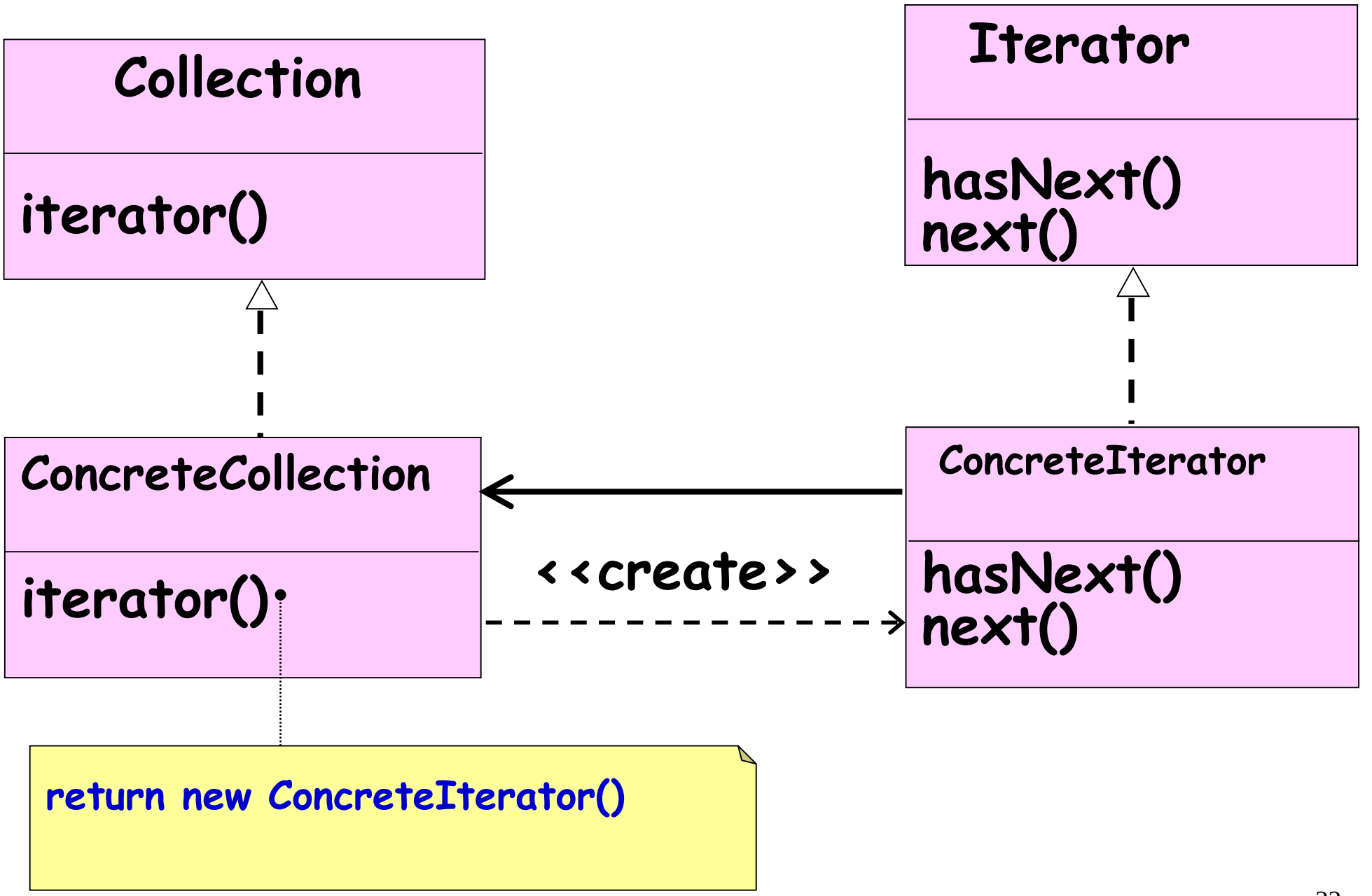
- For vectors this approach does not appear too bad:
  - Because the elements of a vector can be retrieved by specifying their position:

```
v.elementAt(i);
```

Iterating Through a Vector: Using Iterator

```
Iterator  i = v.iterator();
   while (i.hasNext()) {
    Complex c= (Complex)i.next();
      System.out.println(c);
 }
```

Iterator for Java Collection



Iterator in Java

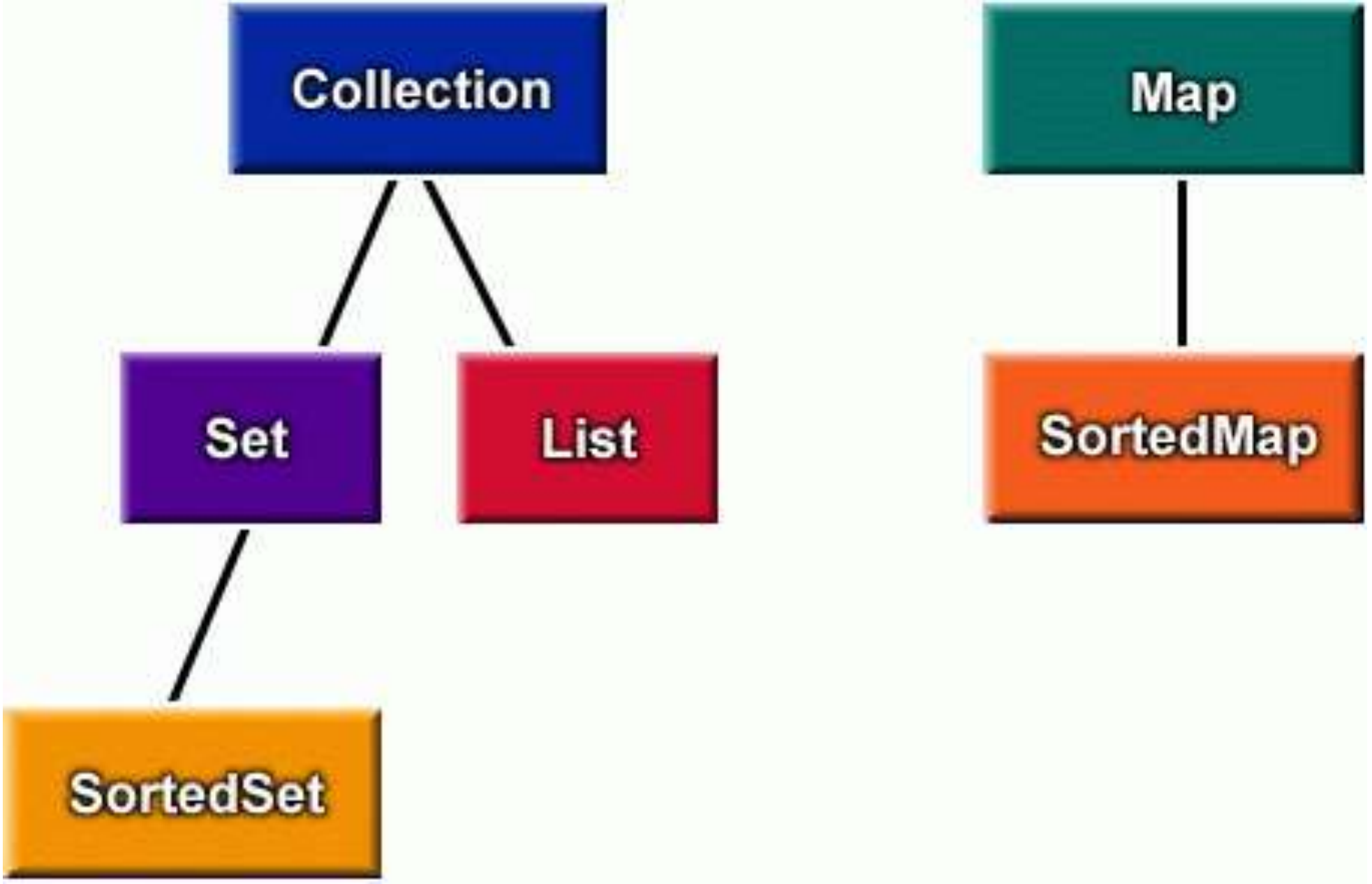
- When we look through the documentation for the Java:
  - Support for the iterator pattern provided in the Java SDK is based on **Iterator interface**:
  - Not abstract class (why?)

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
    public void remove();
}
```

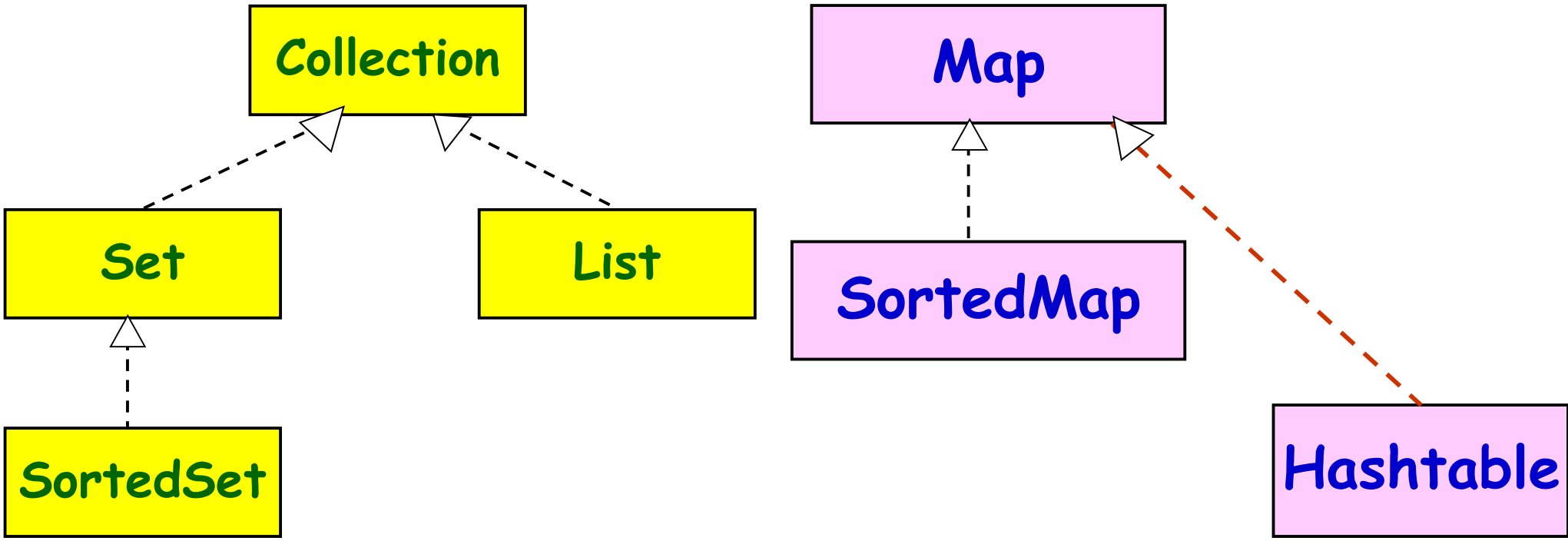
GOF Iterator Pattern in Java

- An Iterator object is an instance of a class that implements the Iterator interface:
  - **Java Iterator interface defines the hasNext(), next(), and remove() methods…**

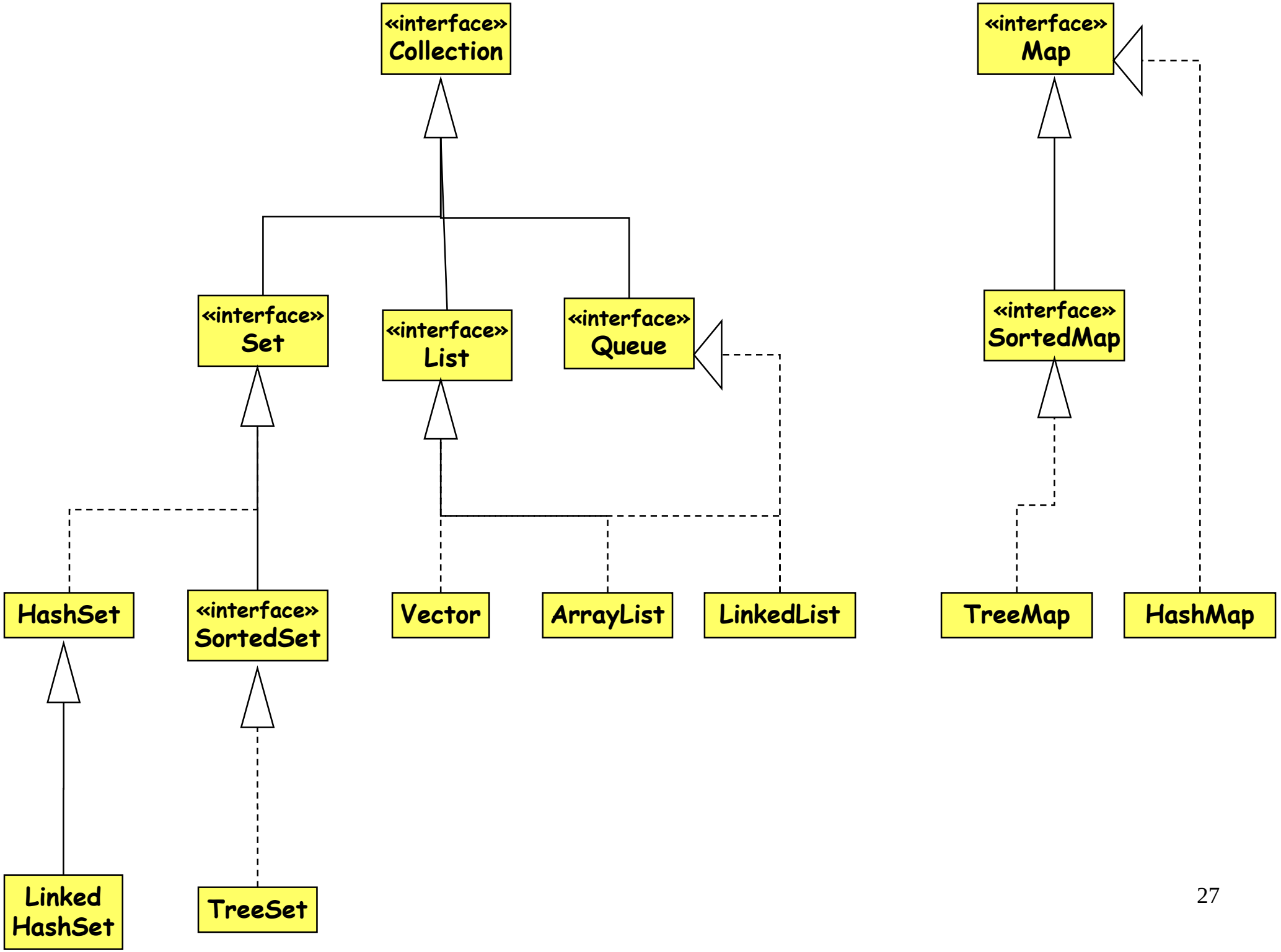
Java Collections



Java Collections



- **Hashtable** is an old (pre-Collections) class
- **Hashtable** has been retrofitted to implement the **Map** interface


«interface»
Collection
«interface»
Map
«interface»
Set
«interface»
List
«interface»
Queue
«interface»
SortedMap
HashSet
«interface»
SortedSet
Vector
ArrayList
LinkedList
TreeMap
HashMap
Linked
HashSet
TreeSet

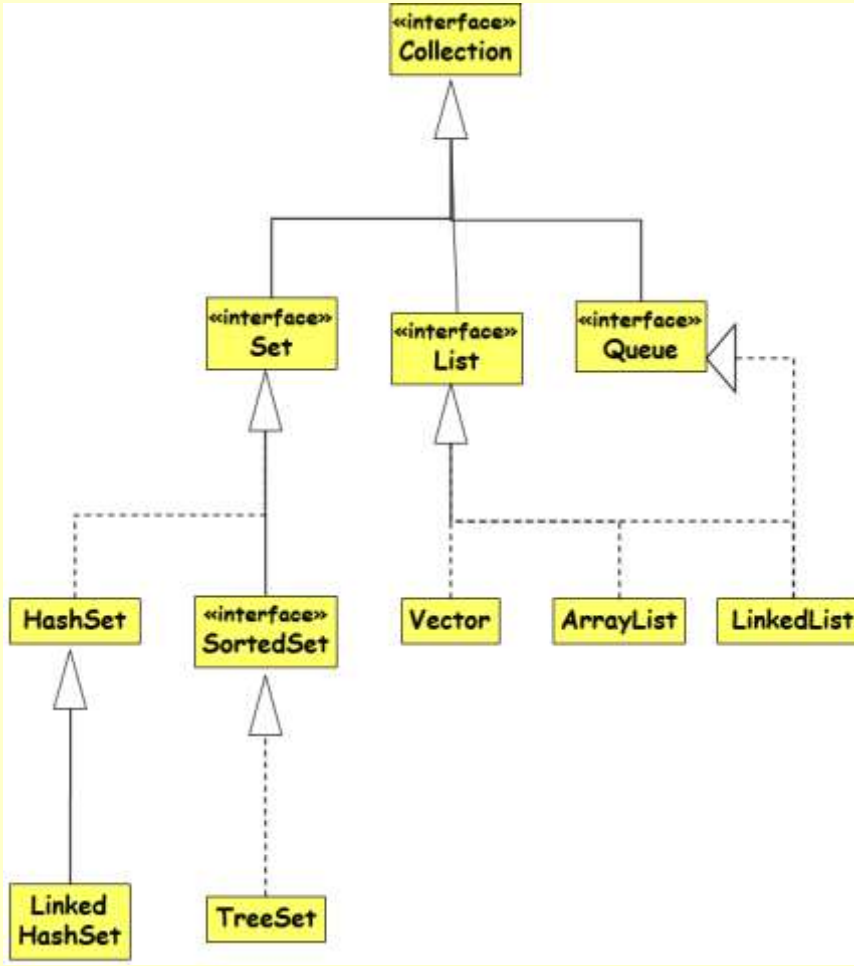
Iterating Through a Hashtable

- **Hashtable does not implement Collection:**
  - Therefore does not have any method to return an Iterator object for the keys and values in the table

- **Instead:**
  - **entrySet() returns the Set of entries**
  - **keySet() returns a Set of the keys in the table**
  - **values() returns a Collection of values in the table**
  - **iterator() can be invoked on these collection objects**

Iterator interface in Java

```
public interface java.util.Iterator {
   public boolean hasNext();
   public Object next();
   public void remove();
}
public interface java.util.Collection {
   ... // List, Set extend Collection
   public Iterator iterator();
}
public interface java.util.Map {
   ...
   public Set keySet();          // keys,values are Collections
   public Collection values();  // (can call iterator() on them)
}
```



The List Interface

- java.util.List interface:

```
public interface List extends Collection {
   Object get(int index);
   Object set(int index, Object element);
   void add(int index, Object element);
   Object remove(int index);
   boolean addAll(int index, Collection c);
   int indexOf(Object o);
   int lastIndexOf(Object);
   ListIterator listIterator();
   ListIterator listIterator(int index);
   List subList(int from, int to);
}
```

- java.util.ListIterator interface:

```
public interface ListIterator extends Iterator {
    boolean hasNext();
    Object next();
    boolean hasPrevious();
    Object previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(Object o);
    void add(Object o);
}
```

The ListIterator Interface

Iterators in Java

- All Java collections have a method **iterator** that returns an iterator for the elements of the collection

```
List list = new ArrayList();
... add some elements ...

for (Iterator itr = list.iterator(); itr.hasNext()) {
  BankAccount ba = (BankAccount)itr.next();
  System.out.println(ba);
}
```

```
set.iterator()

map.keySet().iterator()

map.values().iterator()
```

Adding your Own Iterators

- When implementing your own data structures, it can be very convenient to use Iterators
  - public class PlayerList {
    public int getNumPlayers() { ... }
    public boolean empty() { ... }
    public Player getPlayer(int n) { ... }
    }

Discouraged Nonstandard interface.

  - public class PlayerList {
    public Iterator iterator() { ... }
    public int size() { ... }
    public boolean isEmpty() { ... }
    }

Preferred

Implementing an Iterator

- Three main approaches to create an Iterator for an aggregate class *A*:
  - **Use a separate class**
  - **Use an inner class within *A***
  - **Use an anonymous inner class within *A***

Inner Iterator Class

- Java allows one class to be nested inside another
- Inner class has full access to private data of main class
  - data is still kept as a private field

```
class LinkList {
   private Node head;
   // put all the usual stuff here

   public class LLIterator implements Iterator {
        private Node nextObject;
        public LLIterator() {
            nextObject = head;
        }
   }

   public Iterator iterator() {
   return new LLIterator();
   }
}
```

A Few Issues

- Who controls the iteration?
- **External:**
  - Client explicitly calls all necessary operations for traversal, e.g. client must explicitly request next() on each node
- **Internal:**
  - Client asks the internal iterator an operation to perform, and the iterator applies that operation to every element…

An **external iterator** is active, an **internal** is passive. When the client (i.e. the programmer) controls the iteration, the **iterator** is called **external iterator**. When the **iterator** controls it, it is called an **internal iterator**.

… **Internal** iteration is less error prone and more readable. **But,…**

Pros and Cons

- **External Iterator:**
  - More flexible than Internal.
    - Can compare 2 collections easily
- **Internal Iterator:**
  - Easier to use, as they define the iteration logic for you. Makes portability easier.

Iterator Pattern: Variations

- **Who defines the traversal algorithm?**
  - The iterator is not the only place where the traversal algorithm can be defined.
  - **The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration.** (does not break encapsulation)
  - **This is called a cursor approach**
- A client will invoke the next operation on the aggregate with the cursor as an argument:
  - The next operation will change the state of the cursor.

Multiple Traversal Algorithms

- Complex aggregates may be traversed in many ways.
  - For example, construction of parse tree may require inorder traversal
  - Semantic checking may involve traversing parse tree pre-order or post-order.
  - Code generation may traverse the parse tree post-order.
  - Iterators make it easy to change the traversal algorithm: **Just replace the iterator instance with a different one**

Traversal Issues

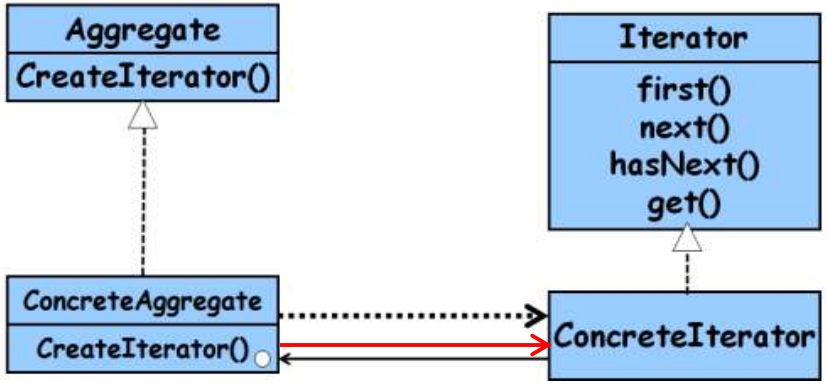
- If the iterator is responsible for the traversal algorithm:
  - It becomes easy to use different iteration algorithms on the same aggregate
  - It is easy to reuse the same algorithm on different aggregates.
- But, the traversal algorithm might need to access private data of the aggregate.:
  - Putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.

Implementation Issues

- **How robust is the iterator?**
  - It can become dangerous to modify an aggregate (elements are added or deleted) while you're traversing it.
  - Why not just make a copy?
  - **A robust iterator ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate.**

Implementation Issues

- There are many ways to implement robust iterators.
  - **Most rely on registering the iterator with the aggregate.**

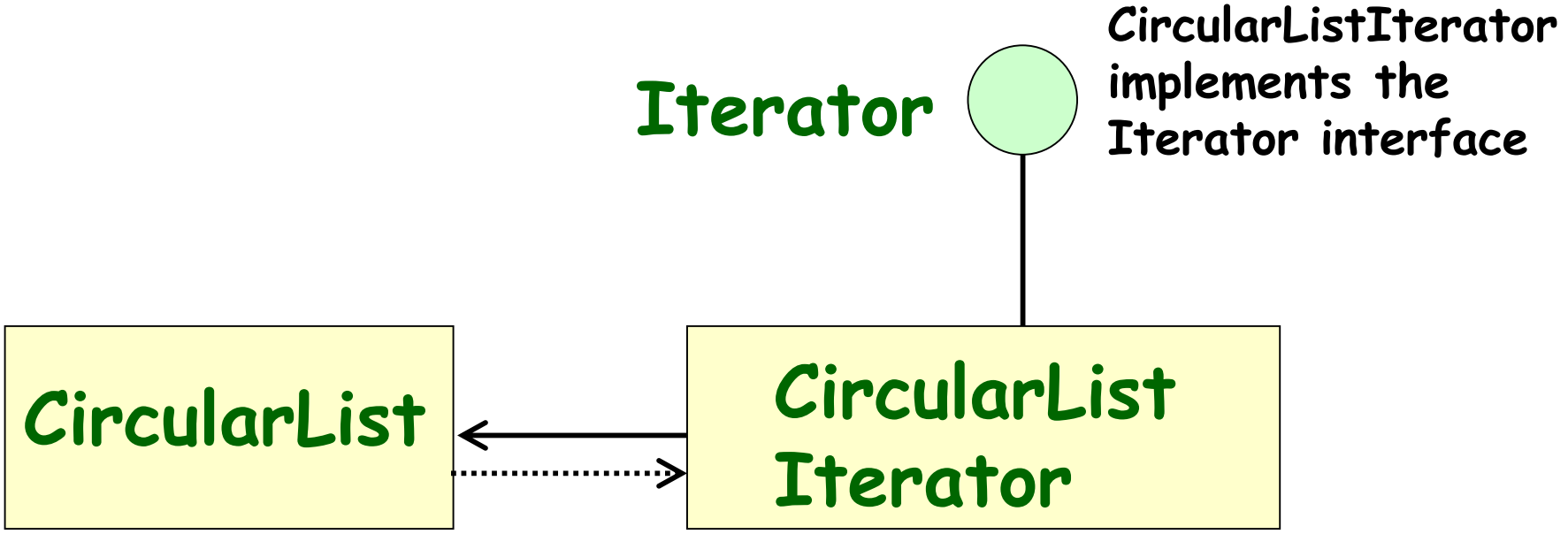


- On insertion or removal:
  - **The aggregate either adjusts the internal state of iterators it has produced,**
  - Or it maintains information internally to ensure proper traversal.

Iterator: Final Analysis

- Programmer gets a consistent interface to traverse any collection…
- It is easy to have multiple iterators traversing the same collection.
- **Flexibility:** An iterator can give access to a filtered view of the items in a collection…

Exercise: Design and Write Code for CircularList Iterator

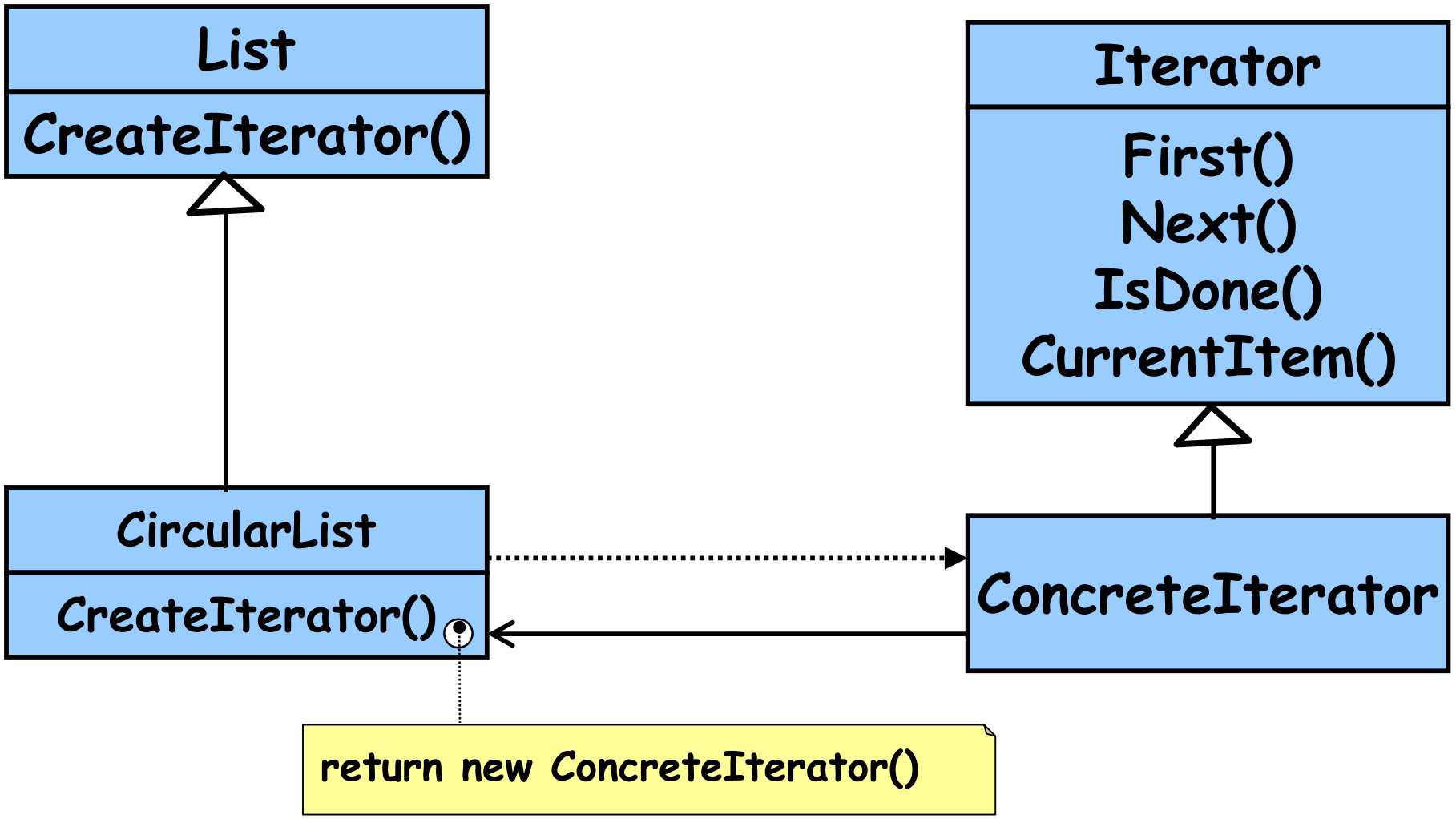


A CircularListIterator object holds a reference to the CircularList object that created it.

Using the CircularList Iterator …

```
CircularList l = new CircularList();

...
//store integers in the list

...
Iterator i = l.iterator();
while (i.hasNext()) {
   System.out.println(i.next());
}
```

Solution: Class Structure



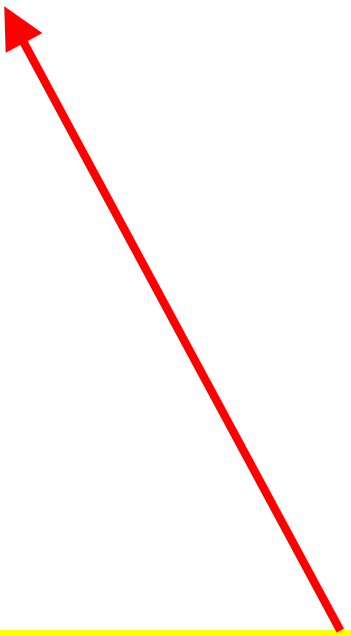
A ConcreteIterator object holds a reference to the CircularList object that created it.

Solution: Class CircularListIterator

```
class CircularListIterator implements
  Iterator {
     public boolean hasNext()    {}
     public Object next()    {}
     public void remove()   {}
}
```

Class CircularListIterator

```
class CircularListIterator implements Iterator{

    private CircularList myList;

    public CircularListIterator(CircularList
       theList) {   myList = theList;  }

    public Object next() {}

    public boolean hasNext() {}

    public void remove() {}

}
```



// Constructor: Pass a reference to the CircularList object to CircularListIterator's constructor

Class CircularListIterator

```
class CircularListIterator implements Iterator {
   private CircularList myList;
                        // constructor not shown…
   private int pos = 0;
   public Object next()  {
      Object o = myList.get(pos);
    // Update pos to indicate the position of the element that
    // will be returned the next time this method is invoked.
      pos++;
      return o;
   }
```

Class CircularListIterator

```
public boolean hasNext() {
    return(pos < myList.count());   }


public void remove()   {
    if (pos > 0) {
        myList.remove(pos-1);
    // This method should throw an
    // IllegalStateException if next() has not
    // yet been invoked, or if remove() has
    // already been invoked after the last
    // invocation of next().
}
```

Modifying Class CircularList

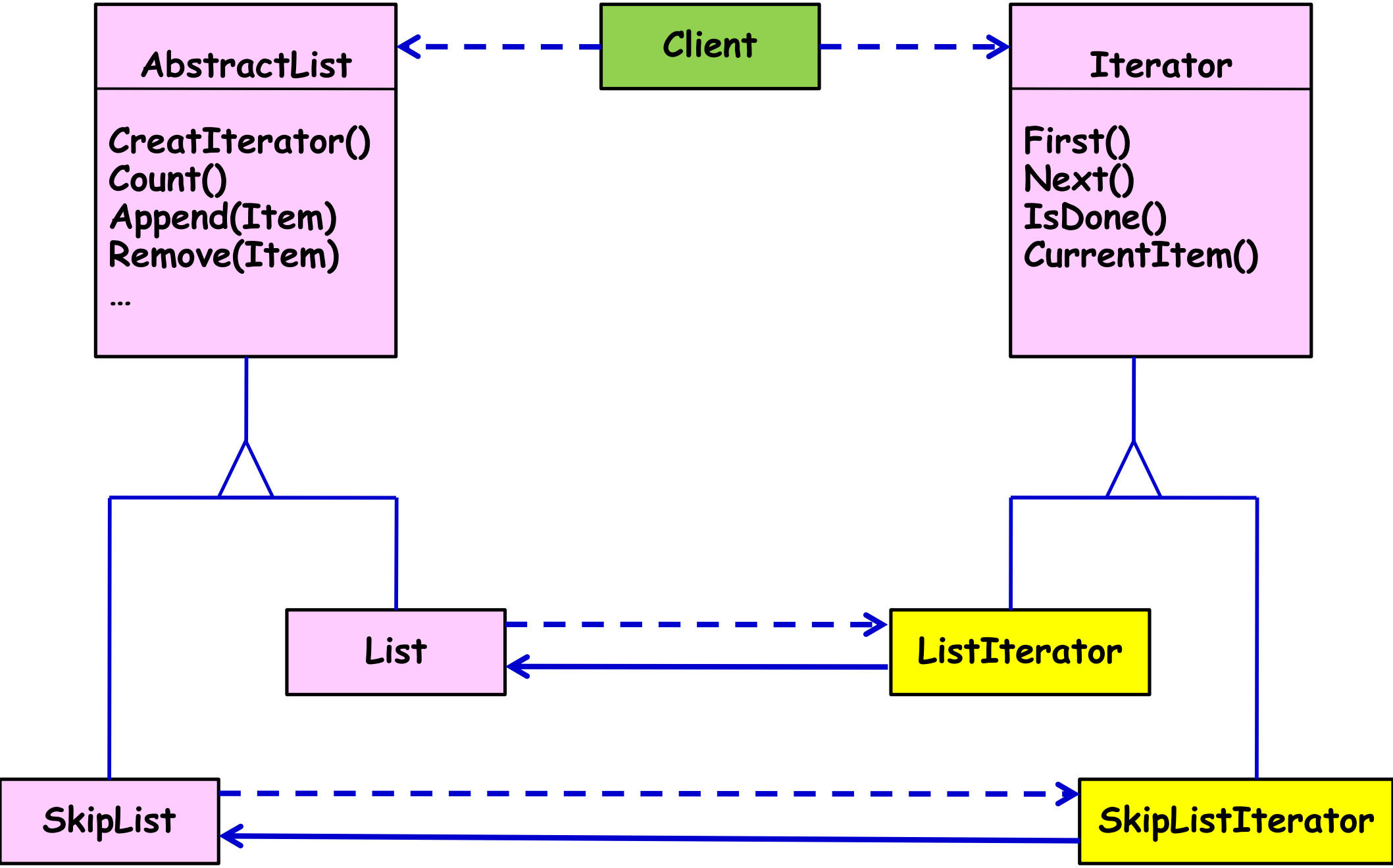
- All that remains now:
  - Define a method in **CircularList** to create and return a **CircularListIterator** object
- In order to be consistent with other Java collection classes:
  - We'll call the method **iterator()**

Modifying Class CircularList

```
class CircularList{

    public Iterator iterator()   {

        // Create a CircularListIterator object,passing
        // it a reference to this CircularList object.
        return new CircularListIterator(this);

    }

}
```

Exercise: Polymorphic Iterators

- Assume that we have an **AbstractList** class.
- Based on this, we have List and SkipList implementations.
- Design Iterators for these different implementations of the AbstractList.


AbstractList
CreatIterator()
Count()
Append(Item)
Remove(Item)
...
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
List
ListIterator
SkipList
SkipListIterator

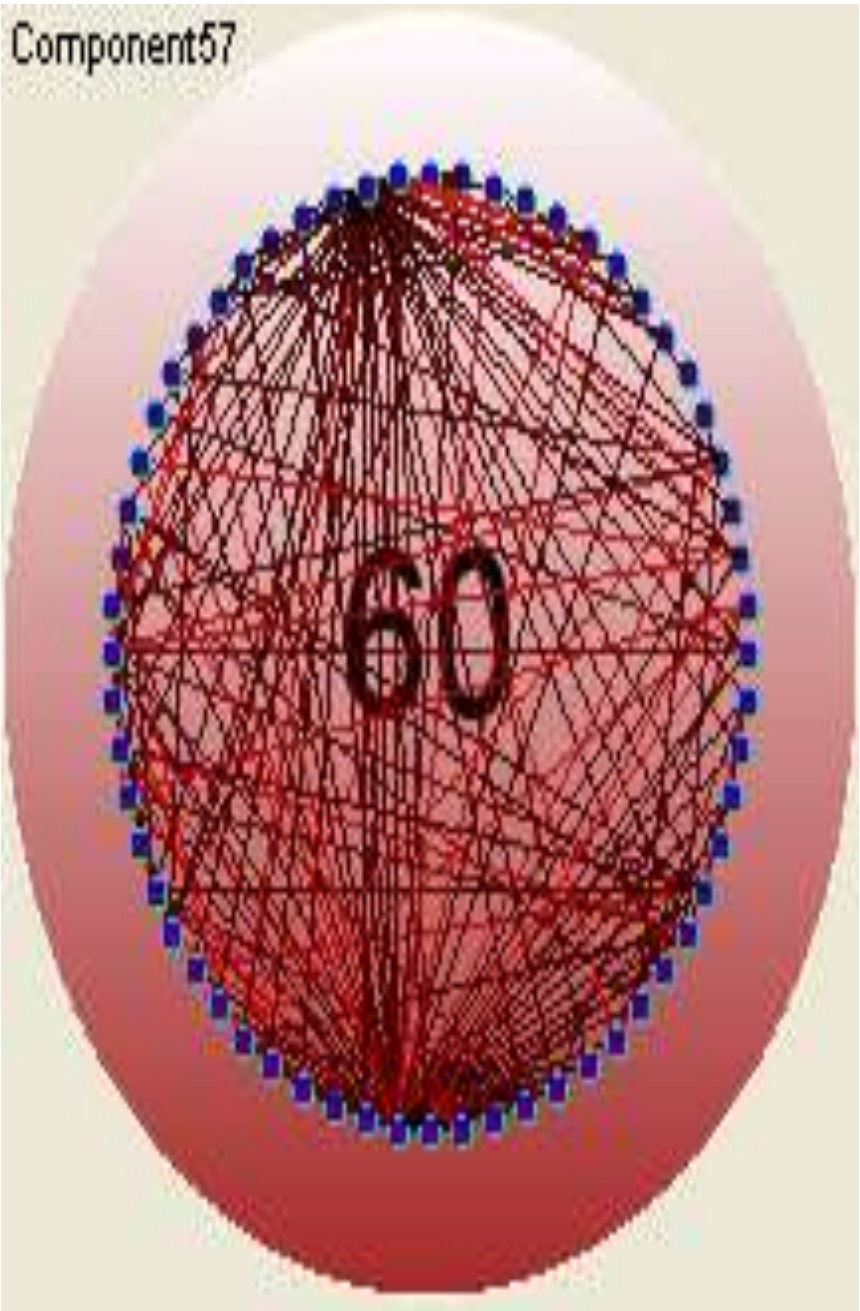
```
List list = new List();

SkipList skipList = new
  SkipList();

Iterator listIterator =
  list.CreateIterator();

Iterator skipListIterator =
  skipList.CreateIterator();

handleList(listIterator);

handleList(skipListIterator);

…
```

```
public void
  handleList(Iterator
  iterator) {

iterator.First();

while (!iterator.IsDone()) {

Object item =
  iterator.CurrentItem();

// Code here to process
  item.

iterator.Next();

}

}
```

Mediator Pattern

Mutual Dependencies among Objects - Mozilla 1.4.1

- Object Oriented Design encourages distribution of behavior among objects.
  - Delegation can result in an object structure with many connections between objects.
- **In the worst case, every object ends up knowing about every other.**

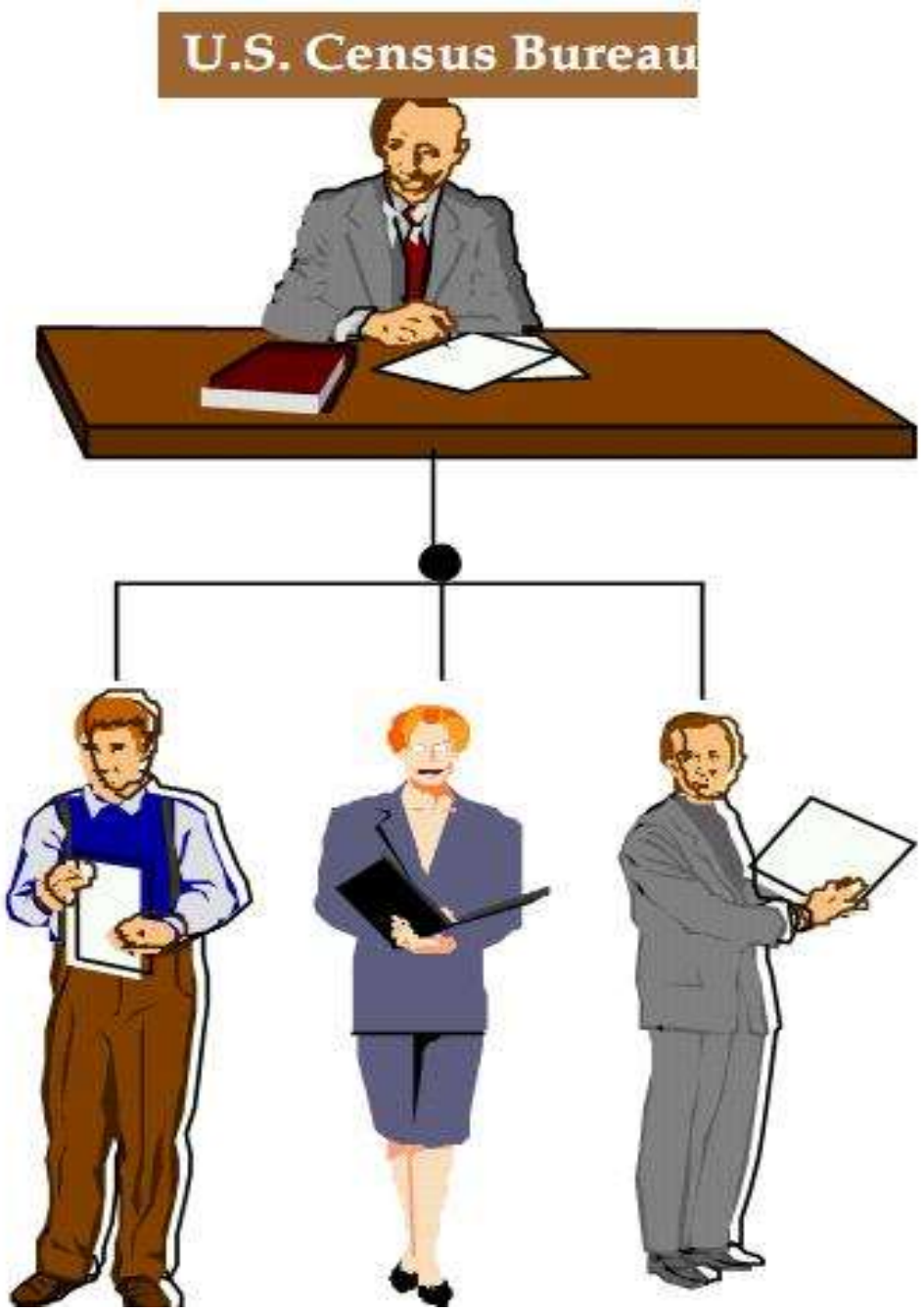


Quote from A Mozilla Developer…

"Even though some of us used to work on Mozilla, we have to admit that the Mozilla code is a gigantic, bloated mess, not to mention slow, and with an internal API so flamboyantly baroque that frankly we can't even comprehend where to begin…"

A non-Software Example: ATC

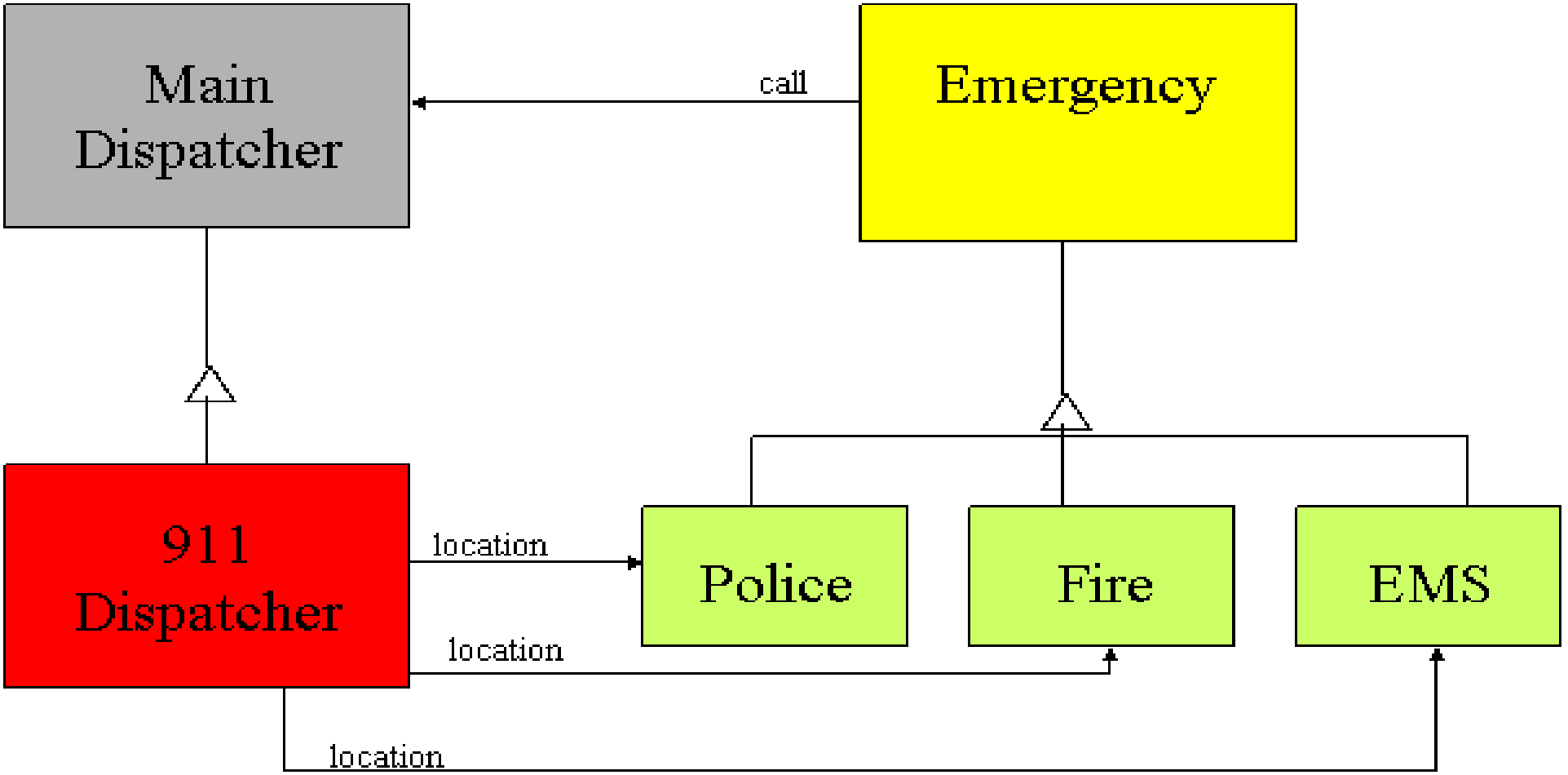
Another Non software example



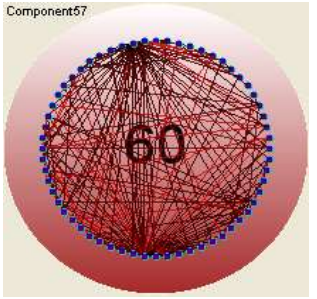
- Master distributes work to identical slaves:
  - Computes a final result from the results that the slaves return.

Yet Another Example

Emergency Dispatch



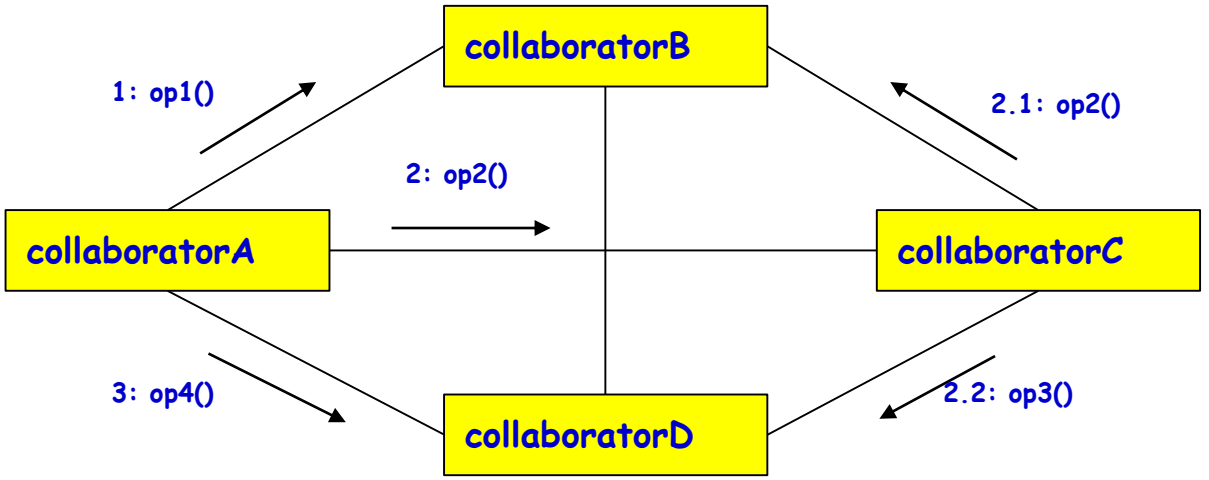
Motivation



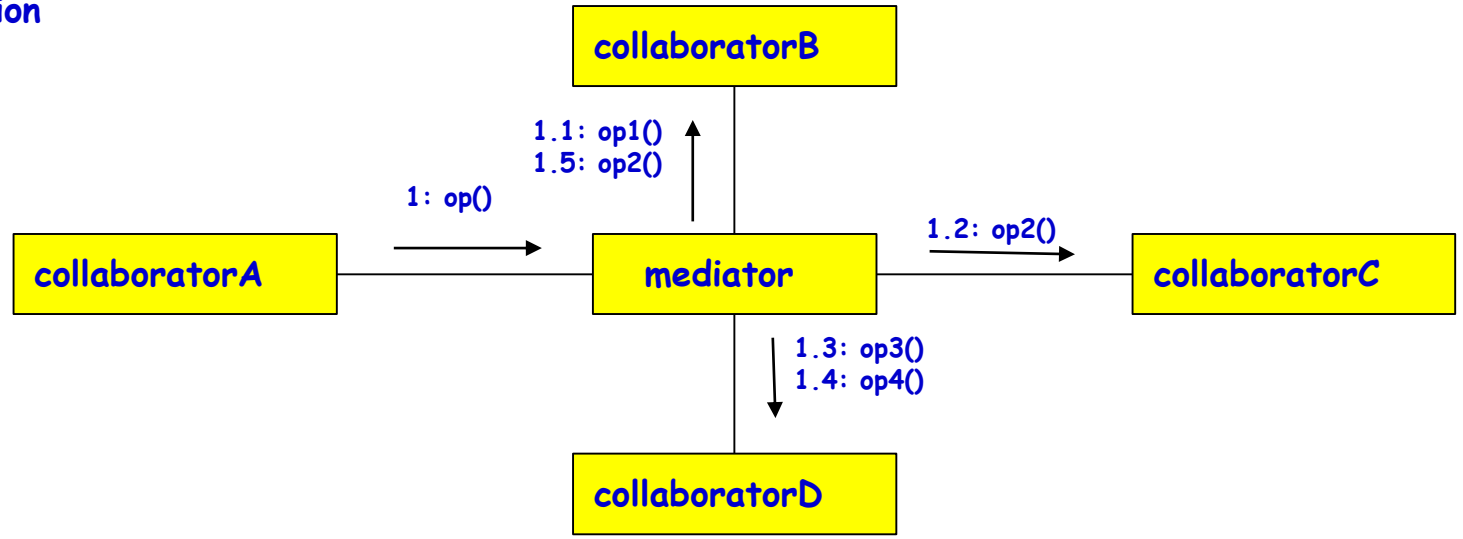
- Object-oriented design leads to distribution of behavior among objects.
  - Leads to many connections between objects.
  - Lots of interconnections make it less likely that an object can work without the support of others.
  - **Sometimes the interactions between objects becomes so intense, that every object in the system ends up knowing about every other object.**

Case for a Mediator!

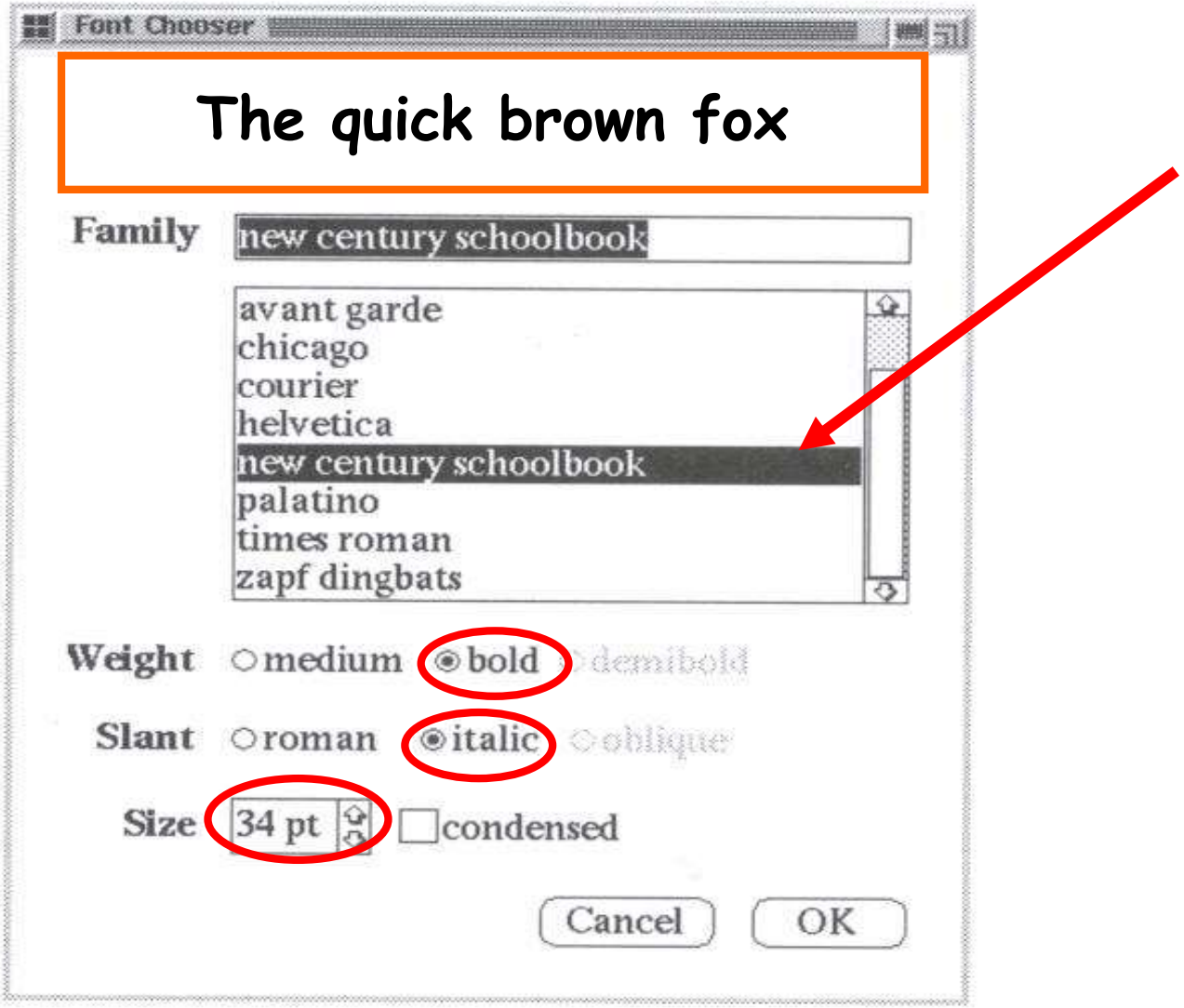
Unmediated Collaboration



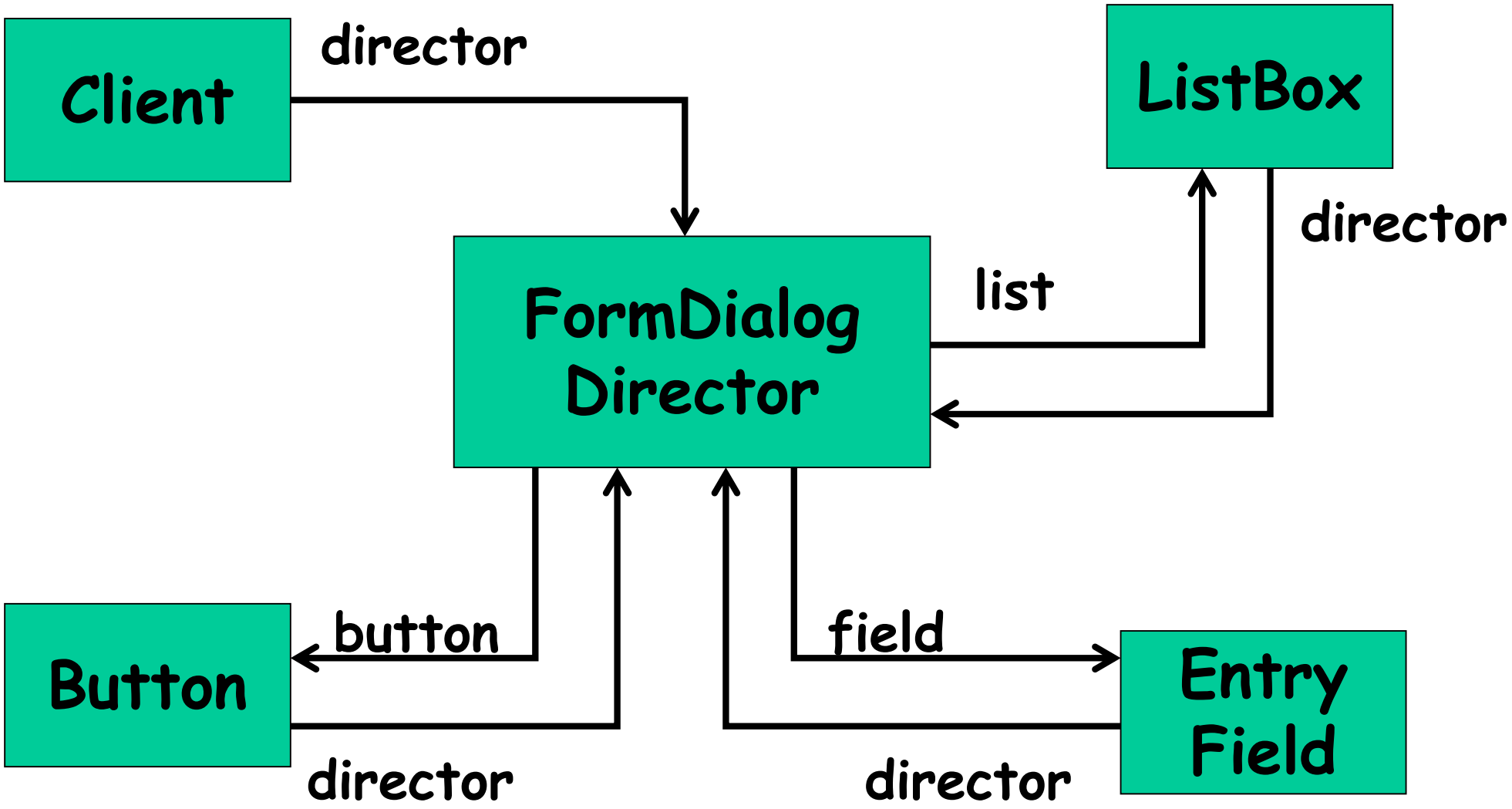
Mediated Collaboration



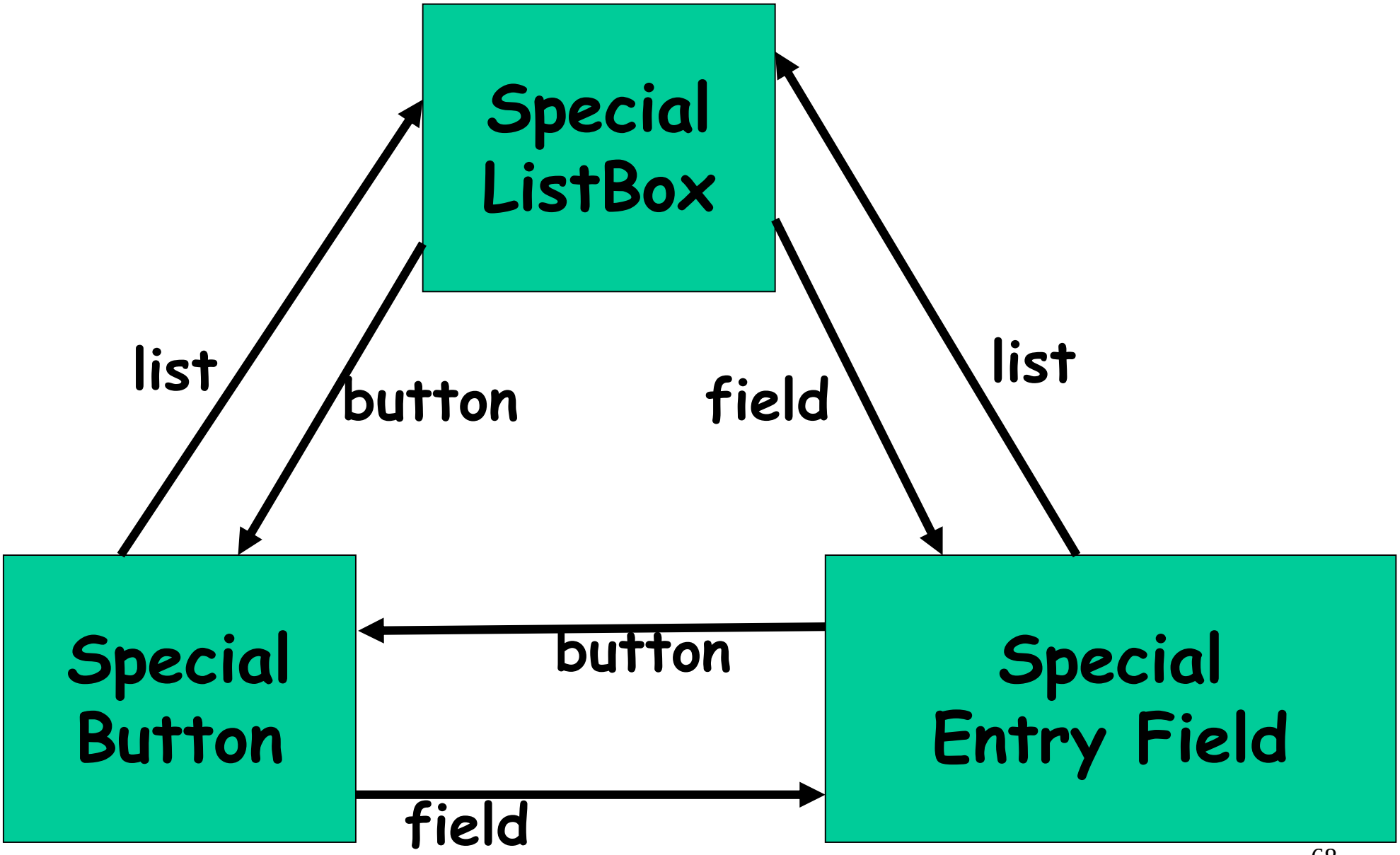
Example- User Interface



Font Editing: Mediator Pattern



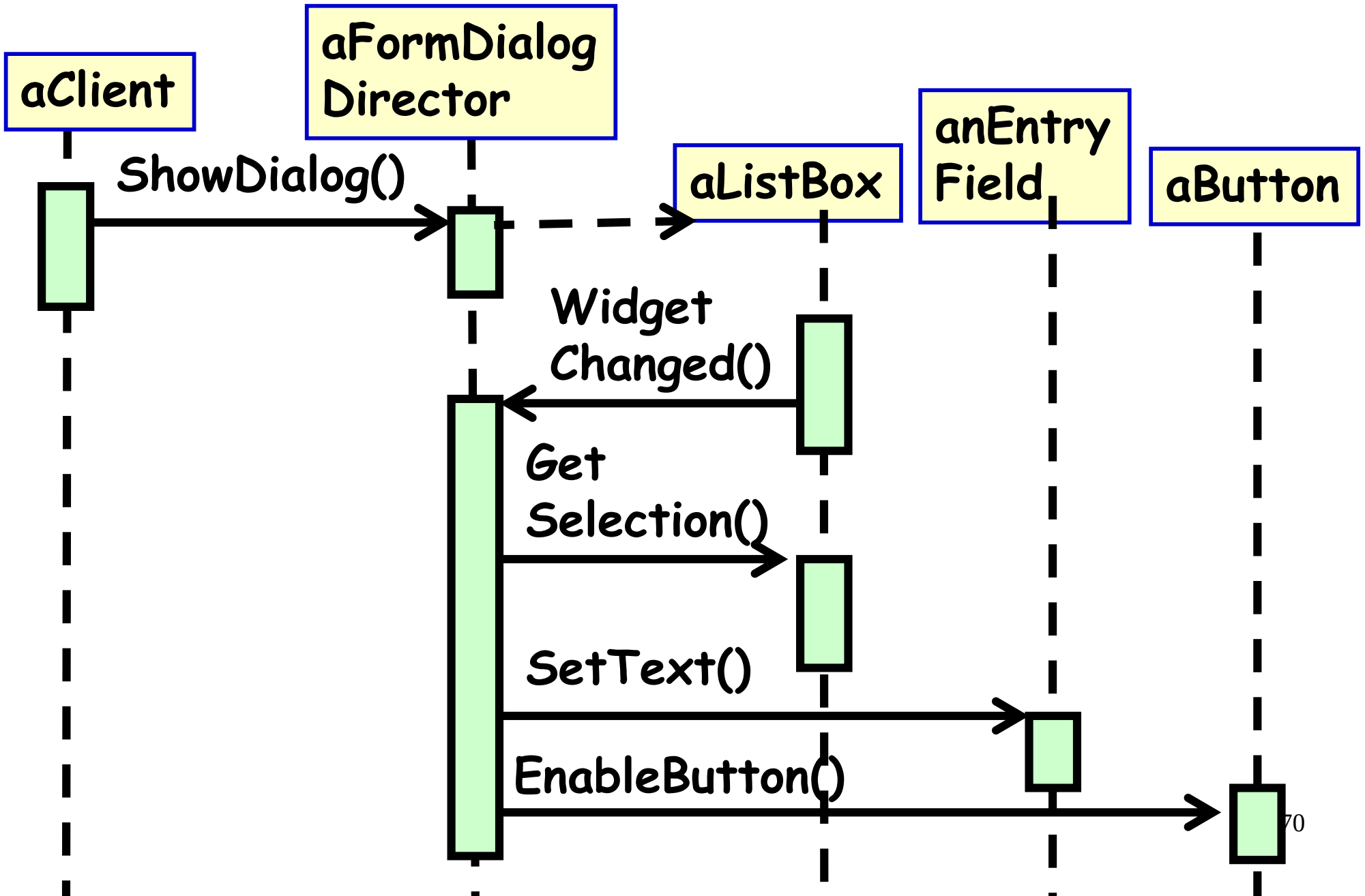
Contrast: Example Coupling Among Classes Without Mediator



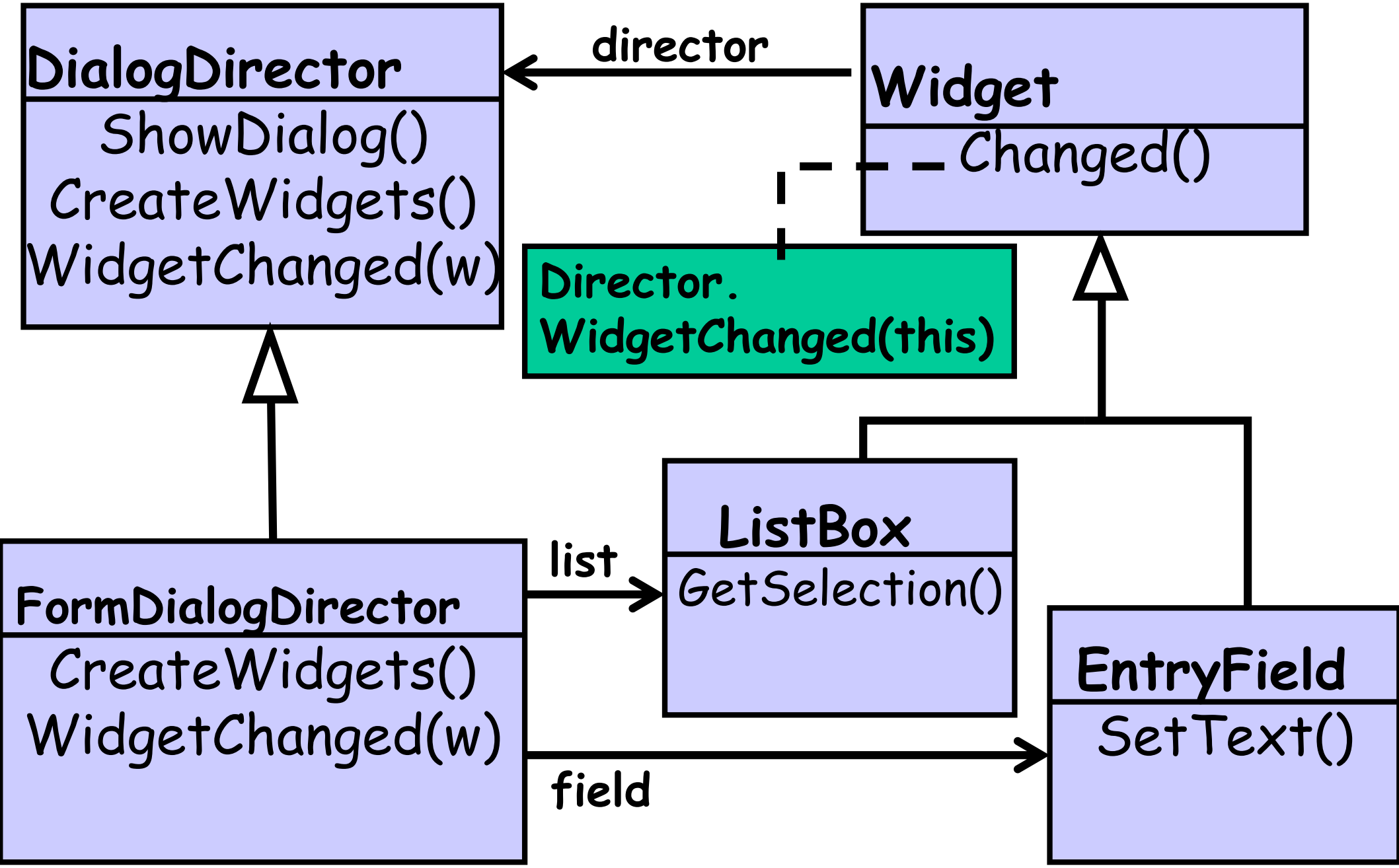
Font Editing: Mediator Pattern

- **FormDialogDirector** acts as the mediator between the widgets in the dialog box.
  - It knows the widgets in a dialog and coordinates their interaction.
  - **It acts as a hub of communications for widgets.**

Mediator Sequence Diagram



Font Editing: Mediator Structure

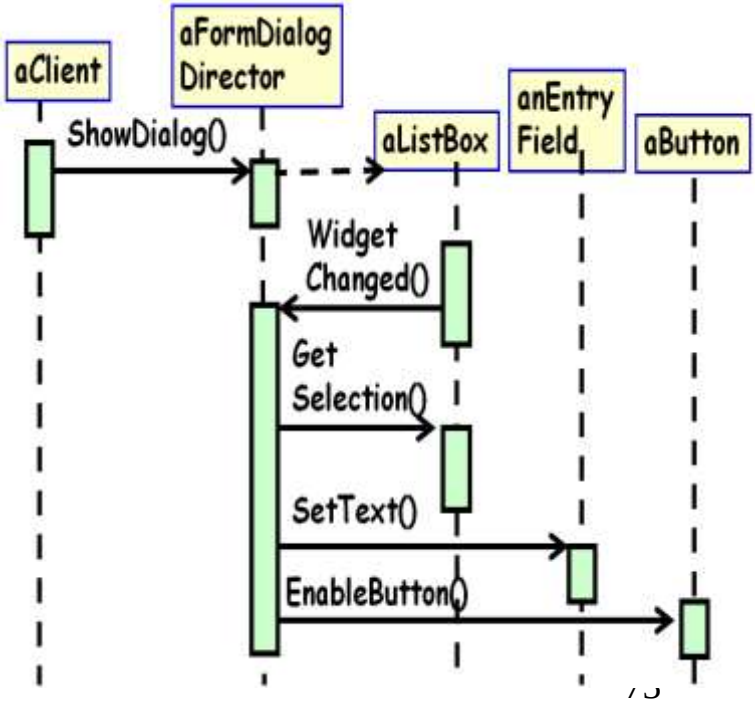


Font Editing: Mediator Pattern

- **DialogDirector:**
  - An abstract class that defines the overall behavior of a dialog.
  - Clients call the ShowDialog operation to display the dialog on the screen.
- **CreateWidgets:**
  - An abstract operation for creating the widgets of a dialog.
- **WidgetChanged:**
  - Another abstract operation, widgets call it to inform their director that they have changed.
- **DialogDirector subclasses:**
  - Override CreateWidgets to create the proper widgets, and they override WidgetChanged to handle the changes.

Font Editing Example -Steps

- List box informs its director that it's changed.
- Director gets the selection from the list box.
- Director passes the selection to the entry field.
- Selected font is displayed.



Font Editing Example: Relationship

- DialogDirector is an abstract class:
  - **Defines the overall behavior of a dialog.**
- Clients call ShowDialog()
- CreateWidgets():
  - Abstract operation for creating widgets.

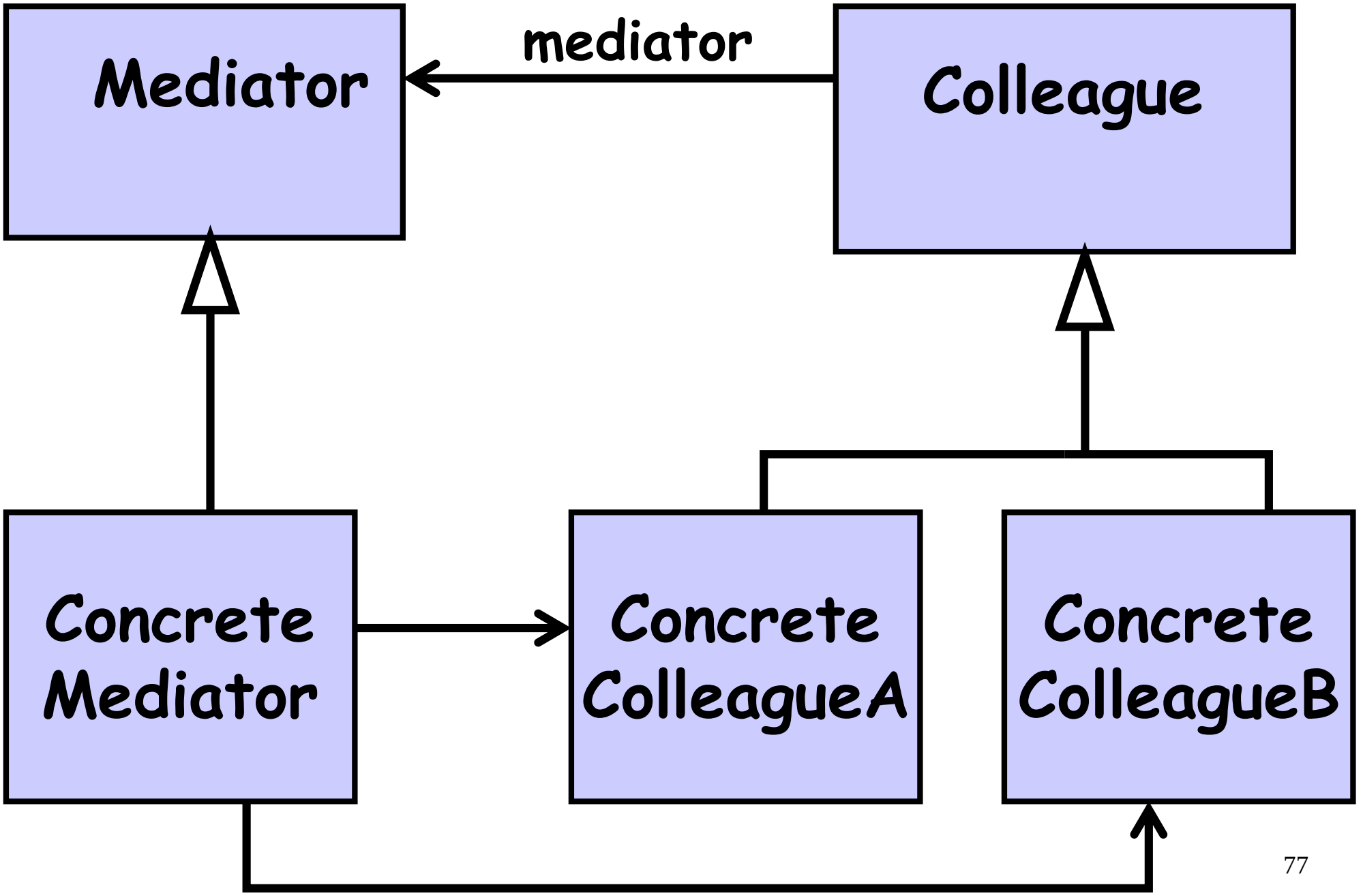
Sample Java Code

```
class FontDialogBox extends Mediator{

private Button ok;   private List fontList;
private TextField fontName;

private CheckBox latino;    private
CheckBox latin2;  //...more


    public FontDialogBox(){
        fontList = new ListBox(this);
        fontName = new TextField(this);
                //...
    }
    public void display(){
            fontList.display();
            fontName.display();      //...
    }
    public void
colleagueChanged(Colleague c){
            if( c==fontList)
                //font_list …
            else if(c==fontName)
                //fontName...
             else if……
    }
            }
}
```

Mediator

- **Encapsulates how a set of objects communicate:**
  - Defines an interface for communicating with Colleague objects
  - Mediator encapsulates the communication.
- **Mediator promotes loose coupling:**
  - The objects only know the Mediator
  - Keeps objects from referring to each other explicitly.

Mediator Pattern: General Structure



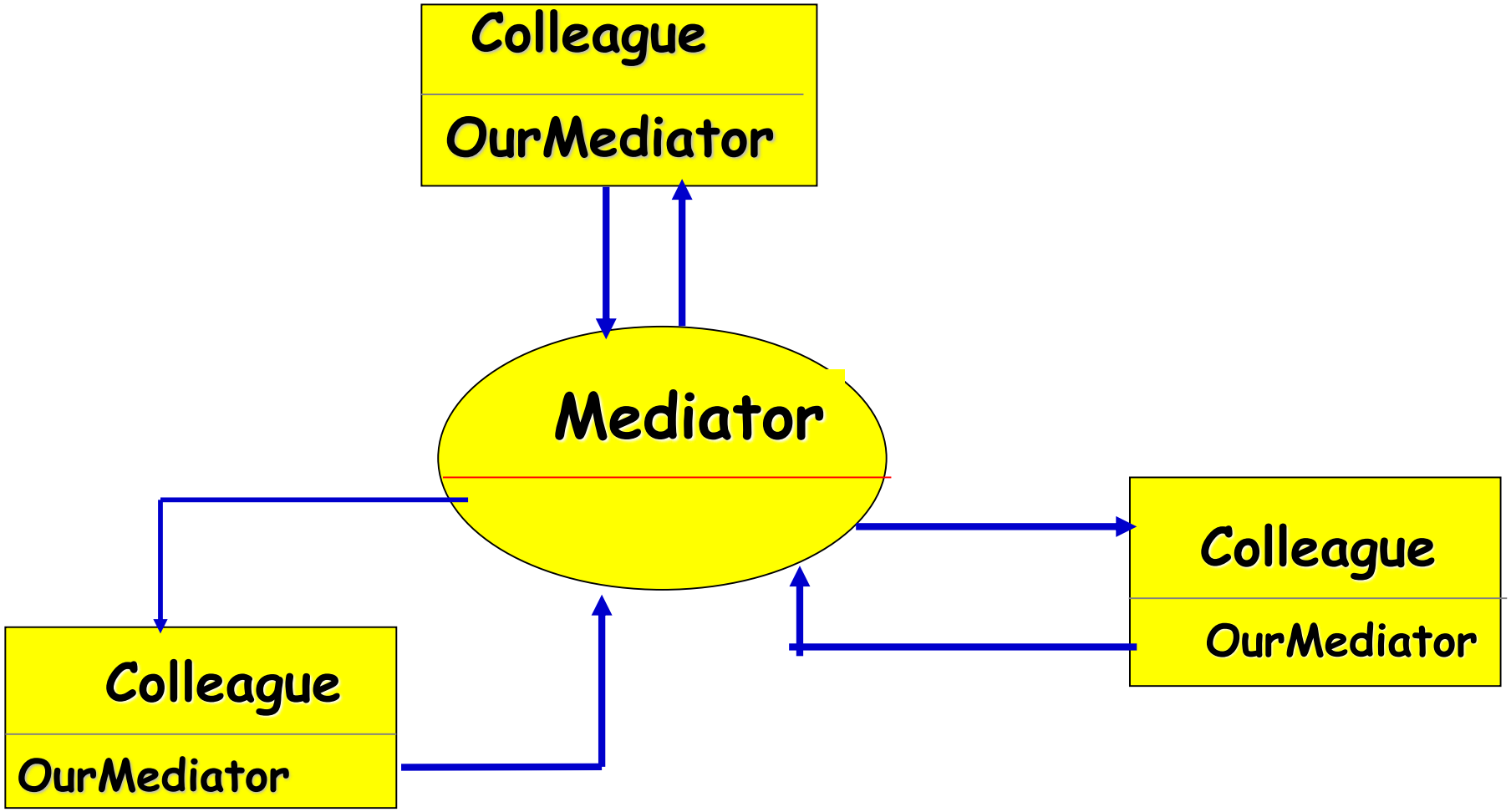
Mediator Pattern Participants

- **Mediator**
  - defines an interface for communicating with Colleague objects.
- **ConcreteMediator**
  - Implements cooperative behavior by coordinating Colleague objects.
- **Colleague classes**
  - Each colleague knows its mediator
  - Each colleague communicates with its Mediator only, it would have otherwise communicated with another colleague

Mediator Pattern Collaborations

- Colleagues send and receive requests from a Mediator object.
- The Mediator facilitates cooperative behavior:
  - Routes requests between the appropriate colleagues
  - **Implements a set of rules for message communication.**

Mediator: Collaboration



Provides a common connection point, centralized behavior management, all with a common interface

Good Points about the Mediator

- **Centralizes control**
  - Less chance of miscommunication.
- **Simplifies object protocols**
  - Replaces many-to-many interactions with one-to-many interactions between mediator and colleagues
  - simple to understand, maintain and extend.
- **Abstracts how objects cooperate**
  - Makes it easier to understand the objects in the system, how they interact and how they are structured.
- **Loose coupling between colleagues promotes Reusability…**

Good Points

- **Limits effects of changes**
  - Localizes behavior that would be otherwise distributed among many objects
  - Changes in behavior require changing only the Mediator class
- **Decouples colleagues**
  - Colleagues become more reusable.

Bad Points about the Mediator

- **Overloaded Mediator:**

  Numerous subclasses of the Mediator and subclasses of those Mediator subclasses... It's just a horrible cycle from this point on...

- **Inefficiency:**

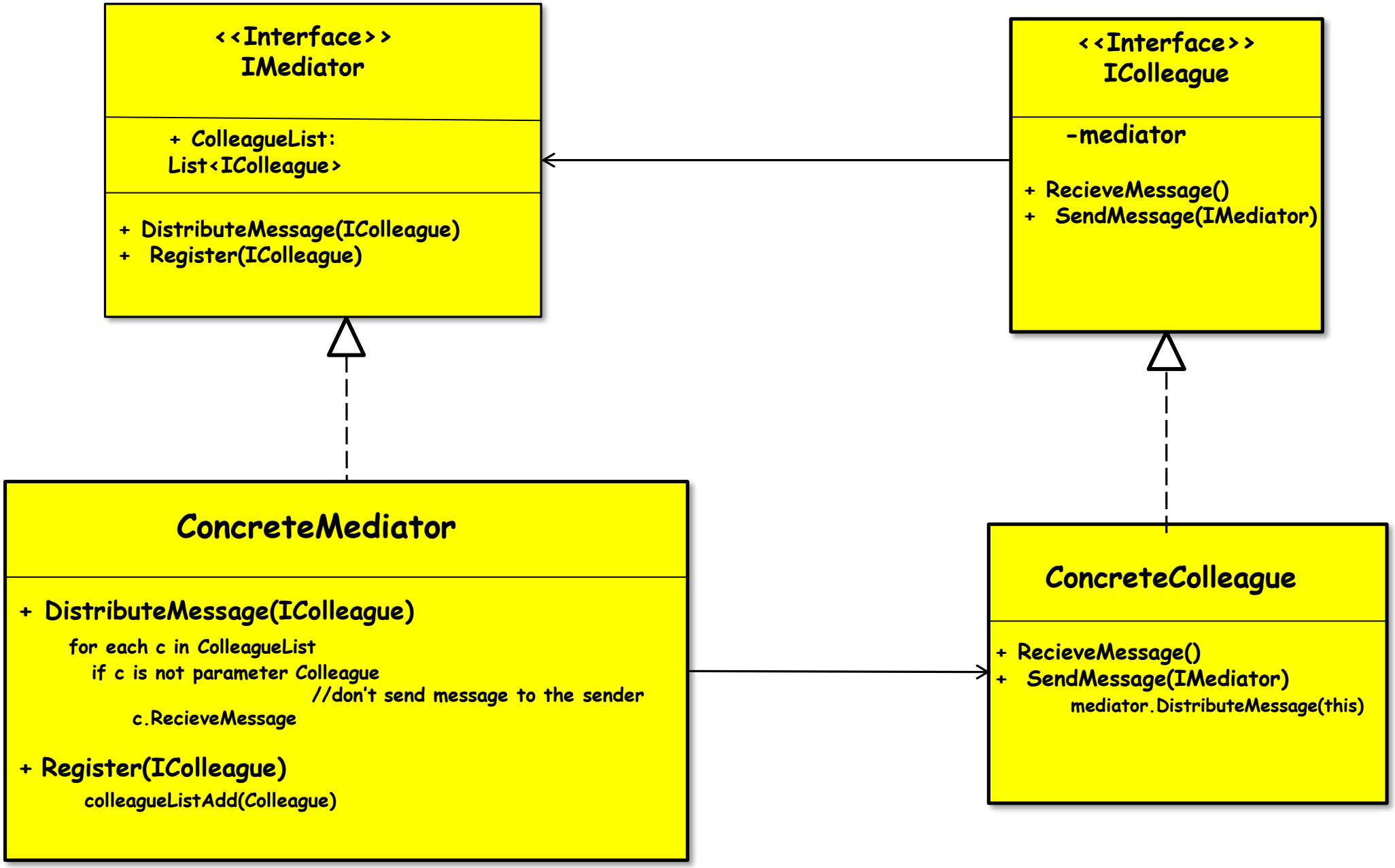
  If there are a relatively small group of objects, it may waste more time as compared to speaking to each other directly.

Might not be a good idea for a relatively small group of objects.

Bad Points cont…

- **Centralizes control:**
  - Mediator can become very complex
  - With complex interactions, extensibility and maintainability may suffer.

Generic Example: Multiple Mediators



```
class MediatorApplicationProgram{
    static void main(string[] args) {
        //list of participants
        IColleague colleagueA = new ConcreteColleague("ColleagueA");
        IColleague colleagueB = new ConcreteColleague("ColleagueB");
        IColleague colleagueC = new ConcreteColleague("ColleagueC");
        IColleague colleagueD = new ConcreteColleague("ColleagueD");
        //first mediator
        IMediator mediator1 = new ConcreteMediator();
        //participants registers to the mediator
        mediator1.Register(colleagueA);
        mediator1.Register(colleagueB);
        mediator1.Register(colleagueC);
        //participantA sends out a message
        colleagueA.SendMessage(mediator1, "MessageX from ColleagueA");
```

Cont…

```
    //second mediator
  IMediator mediator2=new ConcreteMediator();
    //participants registers to the mediator
  mediator2.Register(colleagueB);
  mediator2.Register(colleagueD);
    //participantB sends out a message
  colleagueB.SendMessage(mediator2,
"MessageY from ColleagueB");
  }
}
```

```
public interface IColleague{
    void SendMessage(IMediator mediator, String message);
    void ReceiveMessage(String message);
}
public class ConcreteColleague implements IColleague{
    private string name;
    public ConcreteColleague(string name) {
        this.name = name;  }
    void SendMessage(IMediator mediator, String message)   {
        mediator.DistributeMessage(this, message);    }
    void ReceiveMessage(String message)  {
        Console.WriteLine(this.name + " received " + message);
    }
}
```

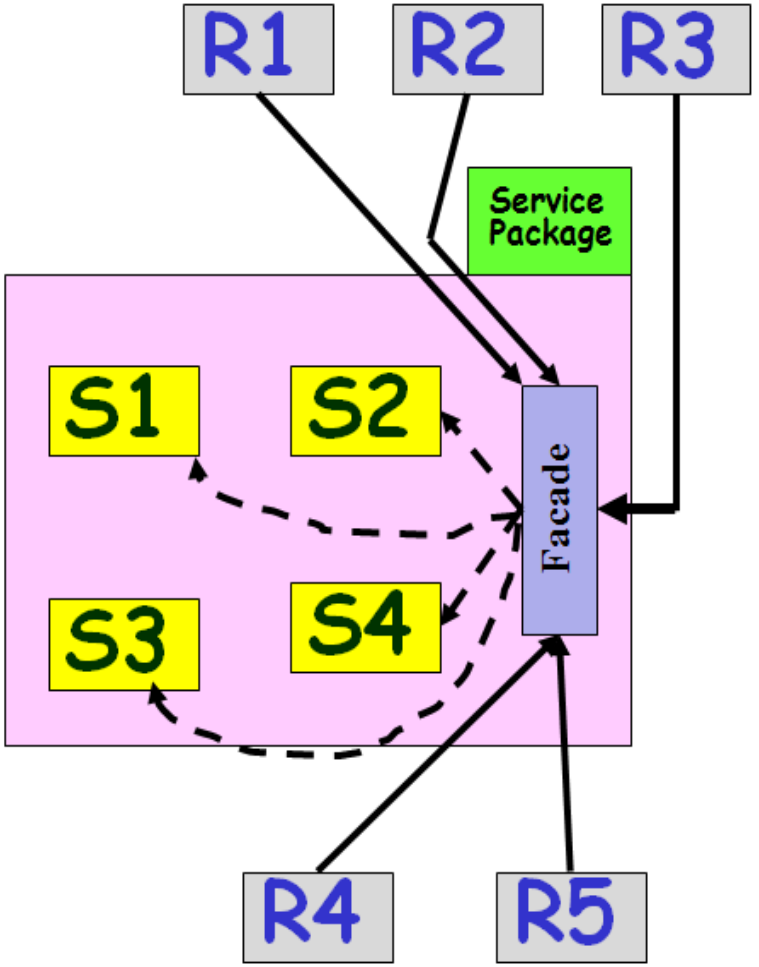
```
public interface IMediator{

   void DistributeMessage(IColleague sender,
   String message);

   void Register(IColleague colleague);

}
```

```
public class ConcreteMediator  implements IMediator{
    private List<icolleague> colleagueList = new List<icolleague>();
    void Register(IColleague<t> colleague) {
        colleagueList.Add(colleague);
    }
    void DistributeMessage(IColleague sender, String message){
        foreach (IColleague c in colleagueList)
            if (c != sender)
              //don't need to send message to sender
                c.ReceiveMessage(message);
    }
}
```

Related Patterns

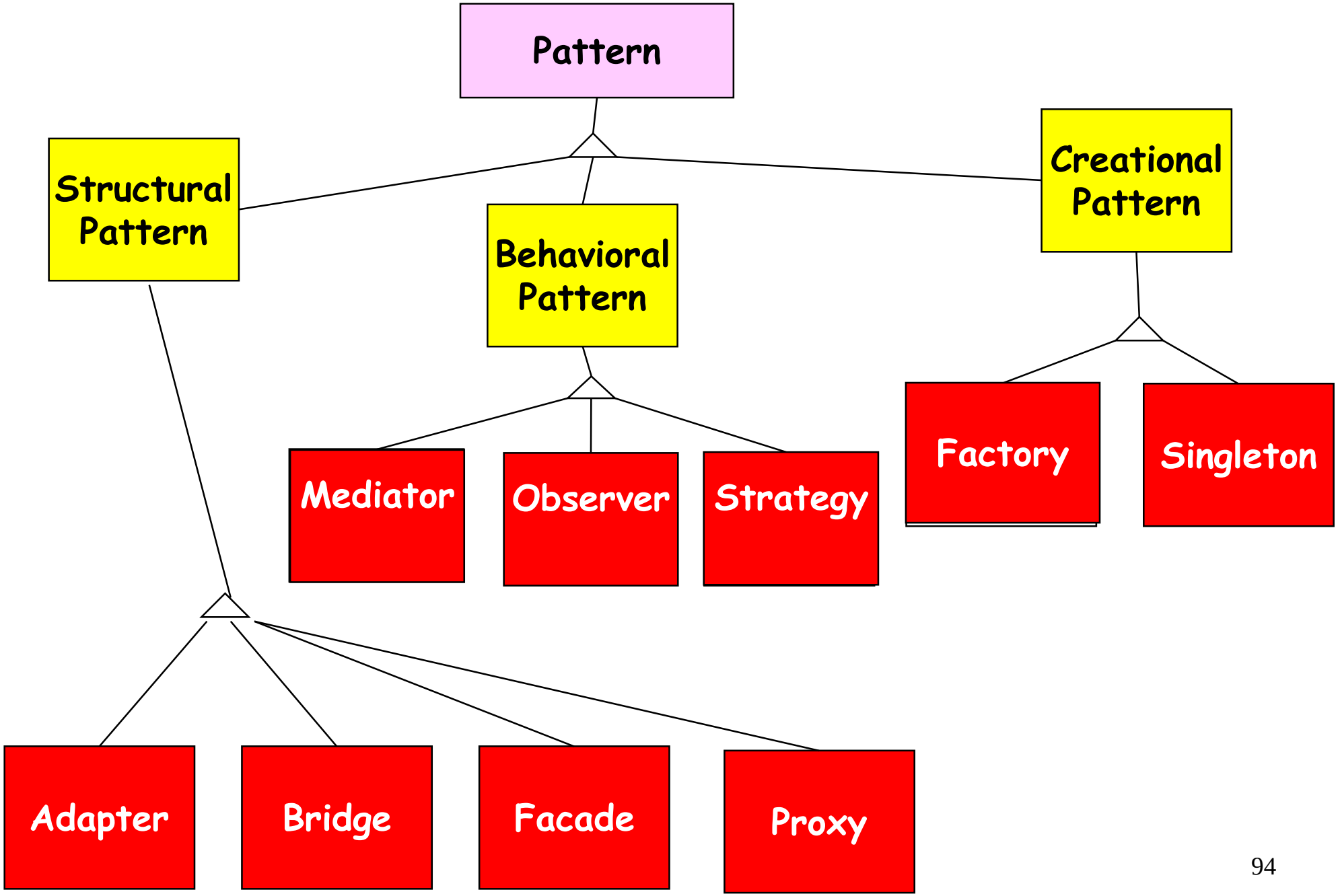
- **Façade**
  - Mediator defines bi-directional interaction but not façade.
  - Facade objects make requests to the subsystem classes but not vice versa.

Façade Pattern



Factory Patterns

Pattern Taxonomy



Factory Patterns

- **Creational patterns:**
  - Make it easier to construct complex objects
  - Hide complexity of creation and eliminate repetitive code.
  - Easy to extend (**Open for extension**)
- **Class creational pattern:**
  - Makes use of inheritance to decide the object to be instantiated
  - **Factory method pattern**
- **Object creational pattern:**
  - Concerns delegation of object instantiation to another object
  - **Abstract factory**

"New" is Closed for Modification

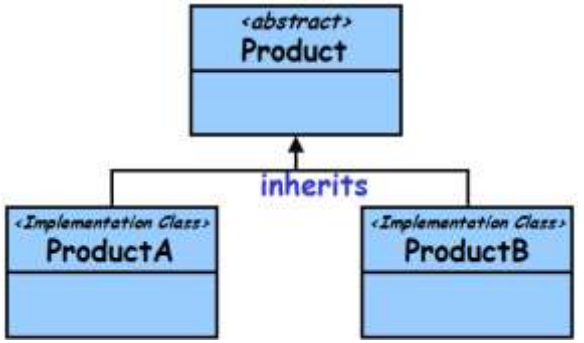
- Use of new operator to directly instantiate a concrete class:
  - **Makes the program closed for modification**
- That may be fine when things are simple, but . . .

Factory Rationale

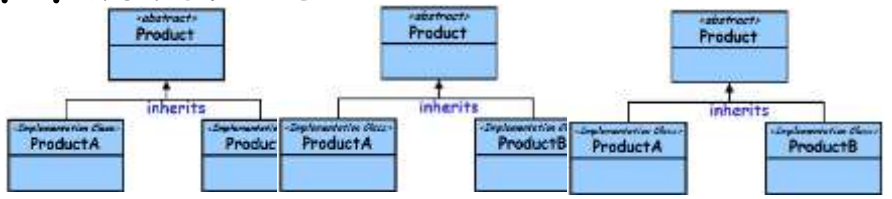
- Why use a Factory?
  - Create new objects without explicitly using the new operator.
  - Can facilitate instantiation of other newly-derived classes.
  - Can initialize and configure the created object.
- **Factories are singletons:**
  - An application typically needs only one instance of a **ConcreteFactory** per product family.

Factory Variations

- Three main Variants:
- **Simple Factory:**
  - Returns an object of a class from a class hierarchy



- **Factory Method pattern:**
  - Produces objects of one type
  - Uses an overridable method to create its objects
  - Subclassed to make new kinds of factories

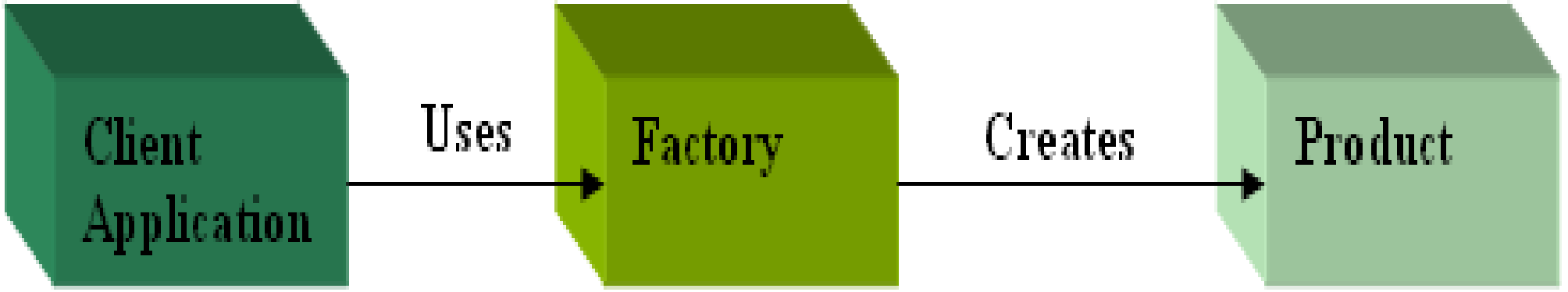


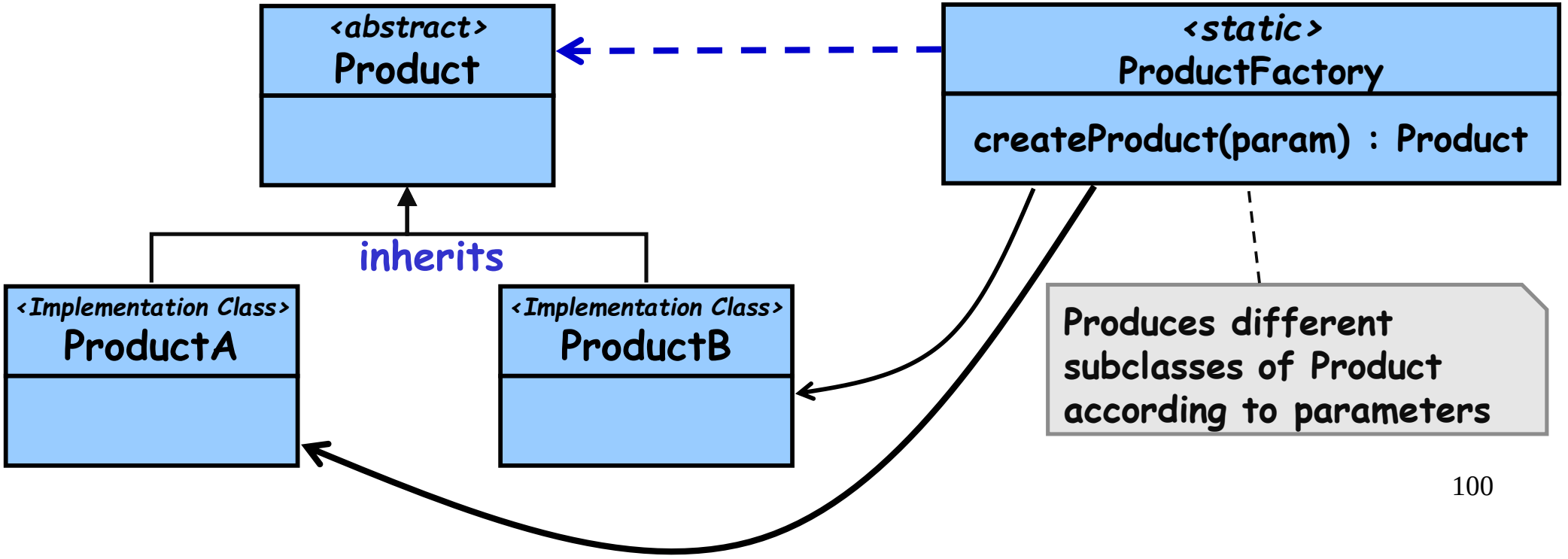
- **Abstract Factory pattern:**
  - Produces objects of many different types (families)

Simple Factory Pattern

- **Factory:**
  - ***A class whose sole job is to easily create and return instances of other classes***
- Instead of calling a constructor:
  - Use a static method in a "factory" class
  - Saves clutter and complexity and improves flexibility
- Examples in Java:
  - **Borders: BorderFactory,**
  - **key strokes: KeyStroke,**
  - **network connections: SocketFactory**

How does Simple Factory Work?





```
Pizza orderPizza() {
  Pizza pizza = new Pizza(); //Base
  pizza.garnish();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

Motivation for Simple Factory

```
Pizza orderPizza(String
  type) {
  Pizza pizza;
  if (type.equals("cheese")) {
  pizza = new CheesePizza();
  } else if (type.equals("greek")) {
  pizza = new GreekPizza();
  } else if
  (type.equals("pepperoni")){
  pizza =new PepperoniPizza(); }
  pizza.garnish();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

```
Pizza orderPizza(String type) {
  Pizza pizza;
  if (type.equals("cheese")) {
    pizza = new CheesePizza();
  } else if (type.equals("greek")) {
    pizza = new GreekPizza();
  } else if (type.equals("pepperoni")) {
    pizza = new PepperoniPizza();
  } else if (type.equals("sausage")) {
    pizza = new SausagePizza();
  } else if (type.equals("veggie")) {
    pizza = new VeggiePizza();
  }
  pizza.prepare();
  pizza.bake();
  pizza.cut();
  pizza.box();
  return pizza;
}
```

Closed for changes!

Encapsulate

Want to introduce new base pizzas…

```
public class SimplePizzaFactory {
  public static Pizza createPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("pepperoni")) {
      pizza = new PepperoniPizza();
    } else if (type.equals("sausage")) {
      pizza = new SausagePizza();
    } else if (type.equals("veggie")) {
      pizza = new VeggiePizza();  }
    return pizza;
  }
}
```

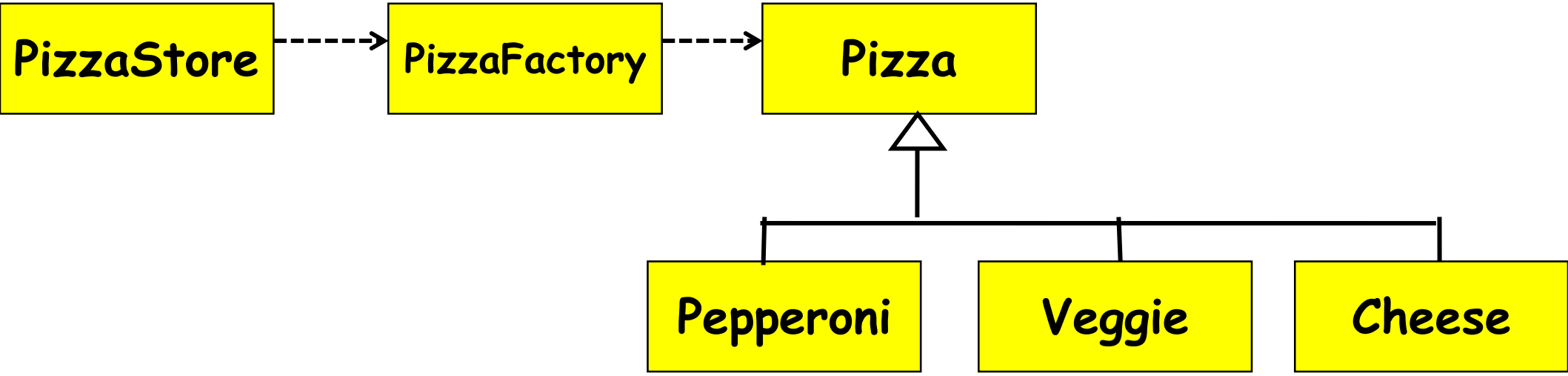
Motivation for Simple Factory

Now orderPizza() would be tidy

```
public class PizzaStore {

  SimplePizzaFactory
  factory;

  public Pizza
  orderPizza(String type) {

    Pizza pizza;

    pizza =
  factory.createPizza(type);

    pizza.garnish();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
  }
}
```

No new operator

Pizza Factory Class Diagram



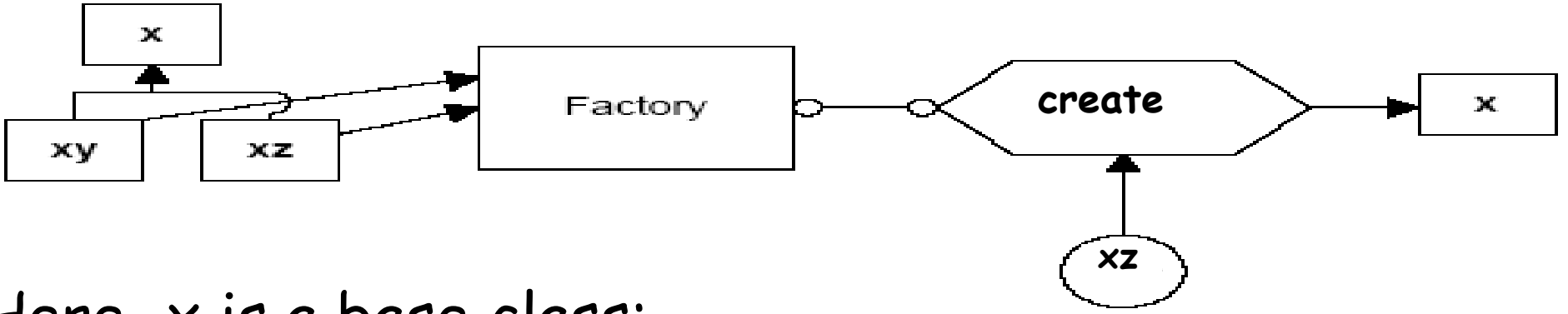
- **A Simple Factory:**
  - Not quite the Factory pattern, later we discuss Factory method abstract **PizzaFactory** class.

Simple Factory: An Explanation

- Pull out the code that builds the instances, and put it into a separate factory class:
  - **Principle:** Identify the aspects of your application that vary and separate them from what stays the same…

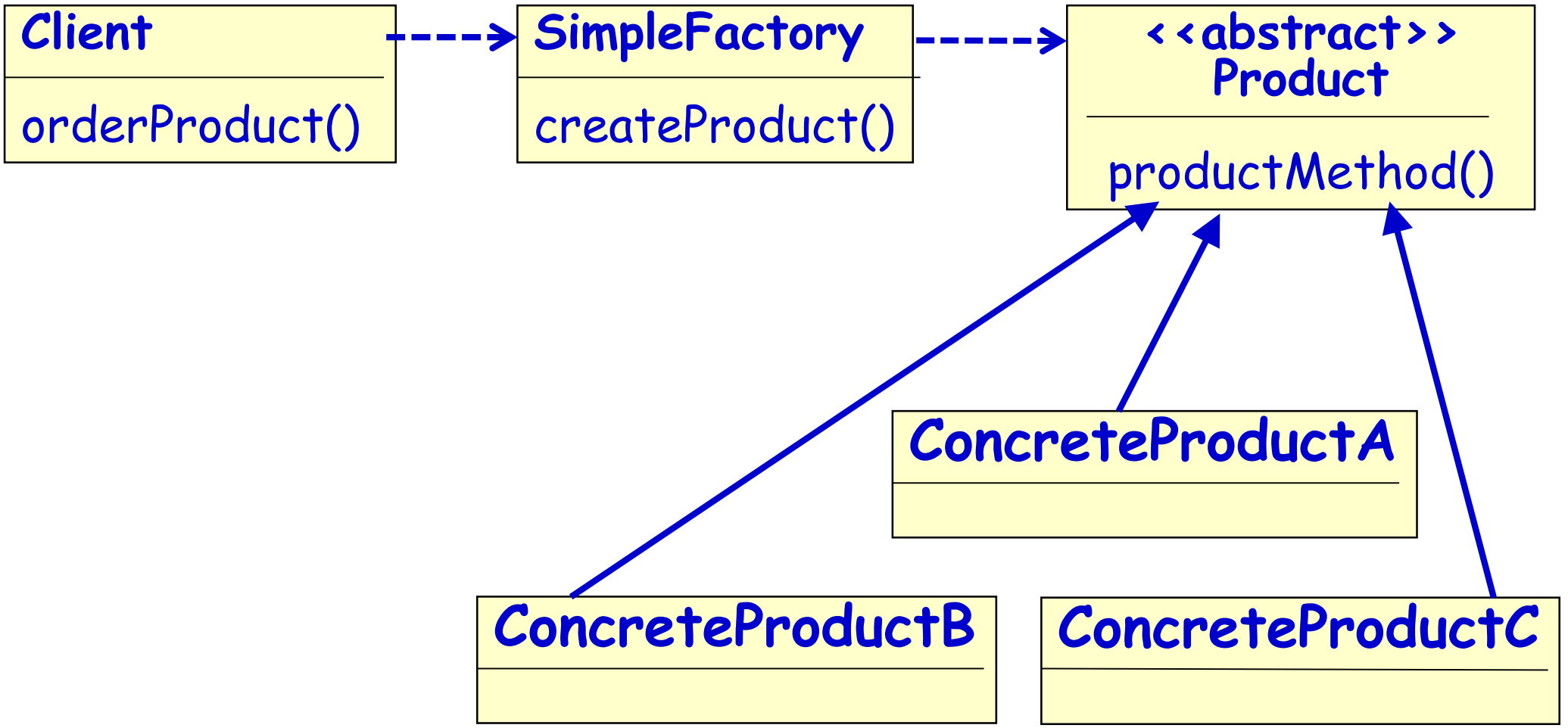
Simple Factory Pattern: Explanation

The simple Factory returns an instance of one of several possible classes depending on the data provided to it.



- Here, x is a base class:
  - Classes xy and xz are derived from it.
  - The Factory class decides which of these subclasses to return depending on the arguments you give it.
- The create() method gets value xz, and returns an instance of the class xz.
  - Which one it returns doesn't matter to the programmer since they are all of type X, but different implementations.

Simple Factory Pattern



Why Would We do This?

- Two main reasons:
  - Ensure consistent object initialization when multiple clients need the same types of objects.
  - Open for modification

Case for Simple Factory: 2 Examples

- Code to construct many GUI components:

```
homestarItem = new JMenuItem("Homestar Runner");
homestarItem.addActionListener(this);
viewMenu.add(homestarItem);
crapItem = new JMenuItem("Crappy");
crapItem.addActionListener(this);
viewMenu.add(crapItem);
```

- Another example (with buttons):

```
button1 = new JButton();
 button1.addActionListener(this);
 button1.setBorderPainted(false);

 button2 = new JButton();
 button2.addActionListener(this);
 button2.setBorderPainted(false);
```

Factory Example 1

```
public class ButtonFactory {
    private ButtonFactory() {}
  public static JButton createButton(
    String text, ActionListener listener, Container
  panel){
    JButton button = new JButton(text);
    button.setMnemonic(text.charAt(0));
    button.addActionListener(listener);
    panel.add(button);
    return button;
  }
}
```

Simple Factory Advantages…

- Creation of buttons etc. by an application object:
  - Avoids significant duplication of code.
  - Makes the client class work at a suitable level of abstraction as these may not be part of the composing object's concerns.

```
public class SimplePizzaFactory {
  public static Pizza createPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if (type.equals("pepperoni")) {
      pizza = new PepperoniPizza();
    } else if (type.equals("sausage")) {
      pizza = new SausagePizza();
    } else if (type.equals("veggie")) {
      pizza = new VeggiePizza();  }
    return pizza;
  }
}
```

Draw Class Diagram for Simple Factory

Simple Factory

Shortcomings?