Design Patterns - II

Object Oriented Software Design

CS2062

Behavioral Patterns

- Identify and realize common communication patterns between objects
- Objective is to increase flexibility in carrying out this communication

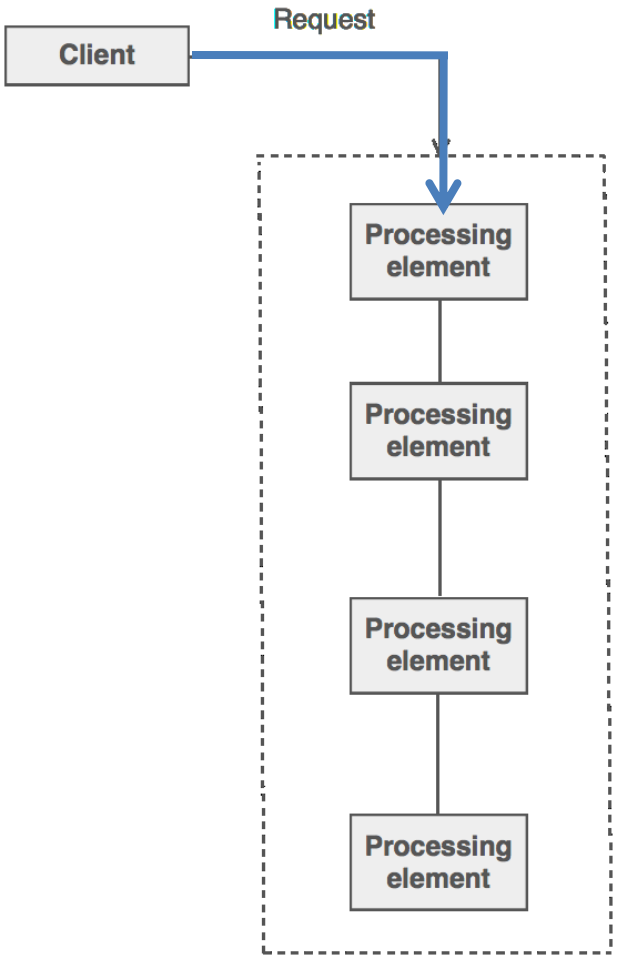
Behavioral Patterns

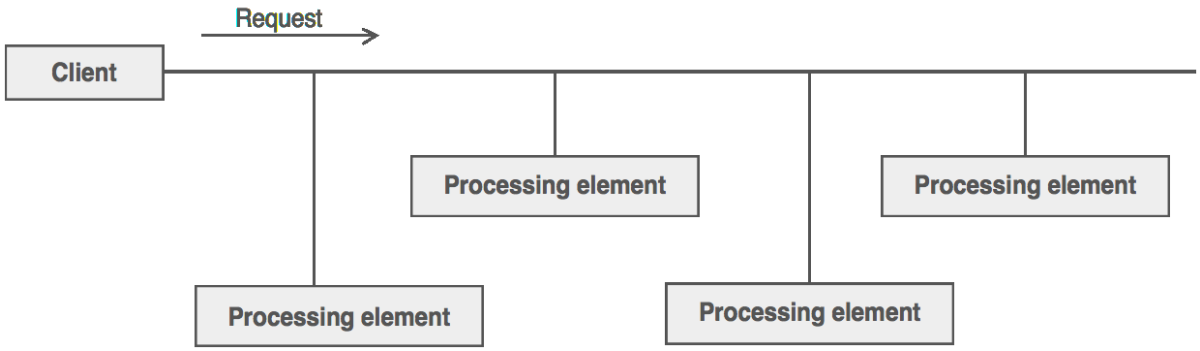
- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- Visitor
- Strategy
- Template Method
- State

Chain of Responsibility

- **Use** to avoid coupling the sender of a request to its receiver
  - i.e. sender does not know who the receiver is
  - Gives more than one object a chance to handle a request
  - Chains the receiving objects and passes the request along the chain until an object handles it

Chain of Responsibility

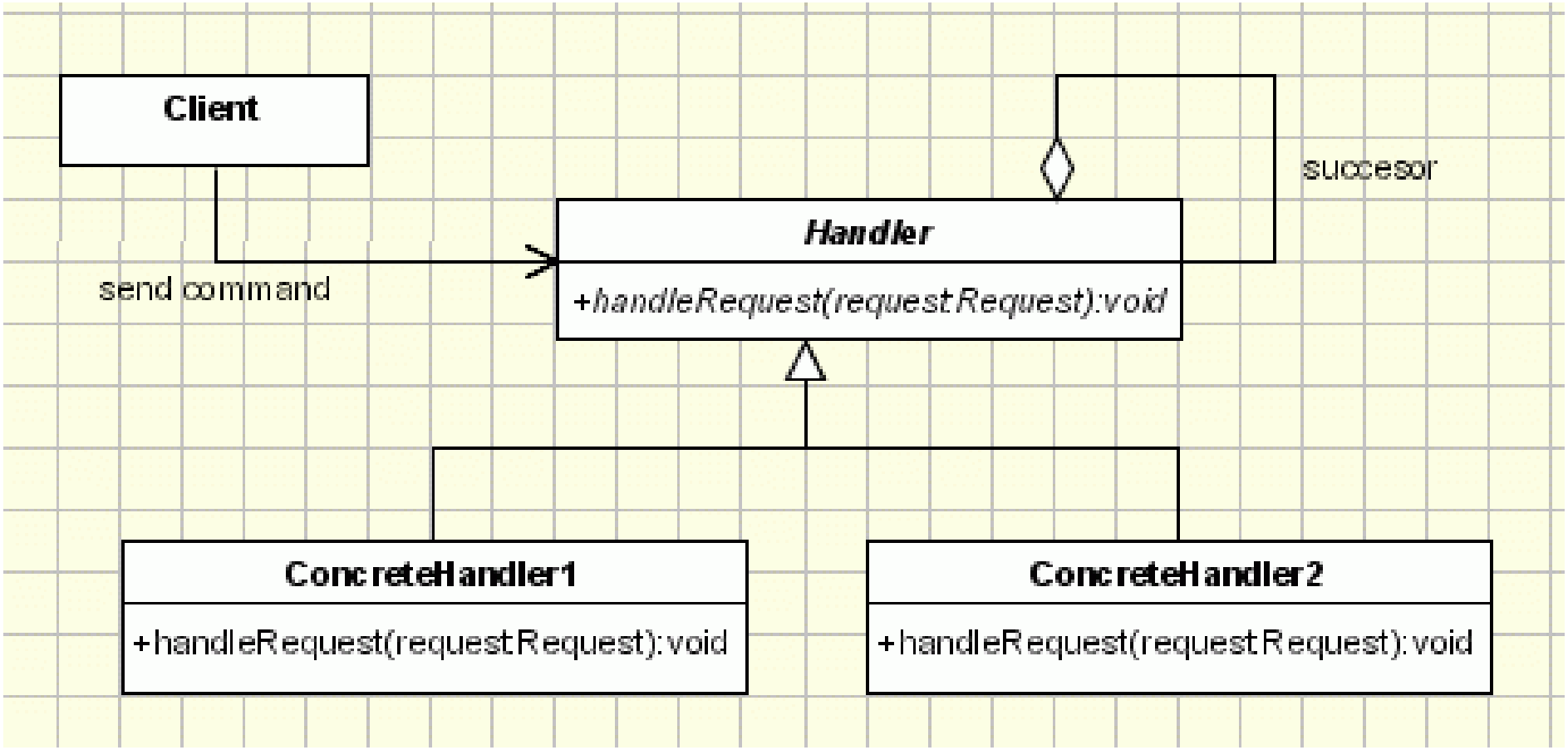




Chain of Responsibility

- Situations where Chain of Responsibility is effective
  - More than one object can handle a command
  - The handler is not known in advance
  - The handler should be determined automatically
  - It’s wished that the request is addressed to a group of objects without explicitly specifying its receiver
  - The group of objects that may handle the command must be specified in a dynamic way

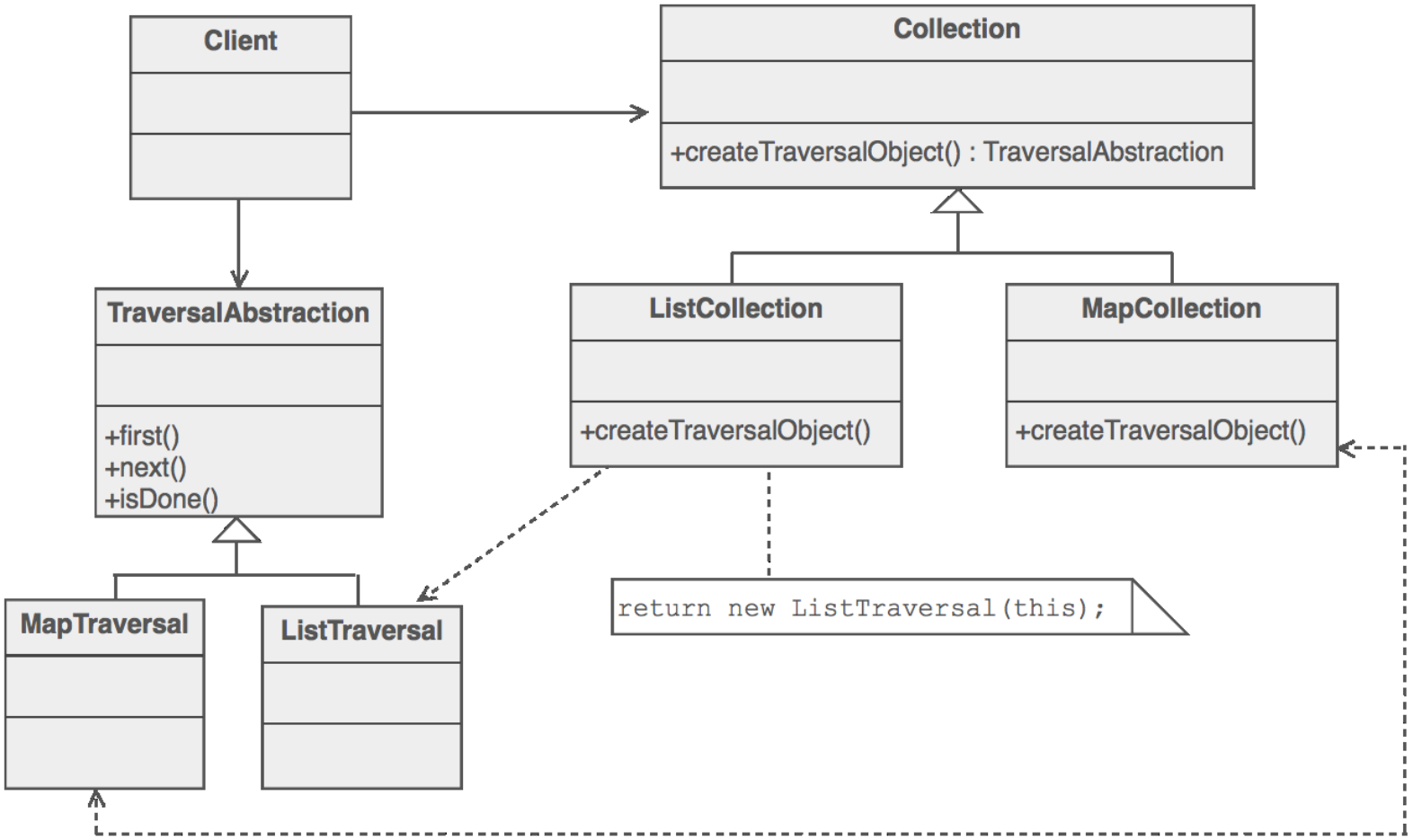
Chain of Responsibility



Iterator

- AKA cursor
- Use when the traversal of different data structures should be done in an abstract way
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Polymorphic traversal

Iterator

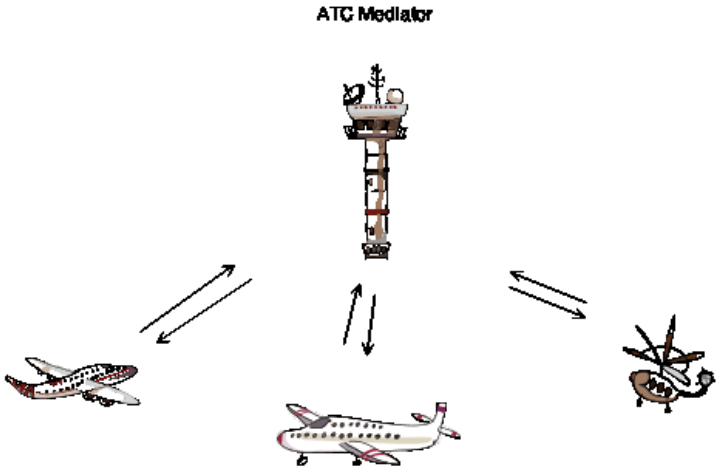

Client
Collection
+createTraversalObject() : TraversalAbstraction
TraversalAbstraction
+first()
+next()
+isDone()
ListCollection
+createTraversalObject()
MapCollection
+createTraversalObject()
MapTraversal
ListTraversal
return new ListTraversal(this);

Iterator

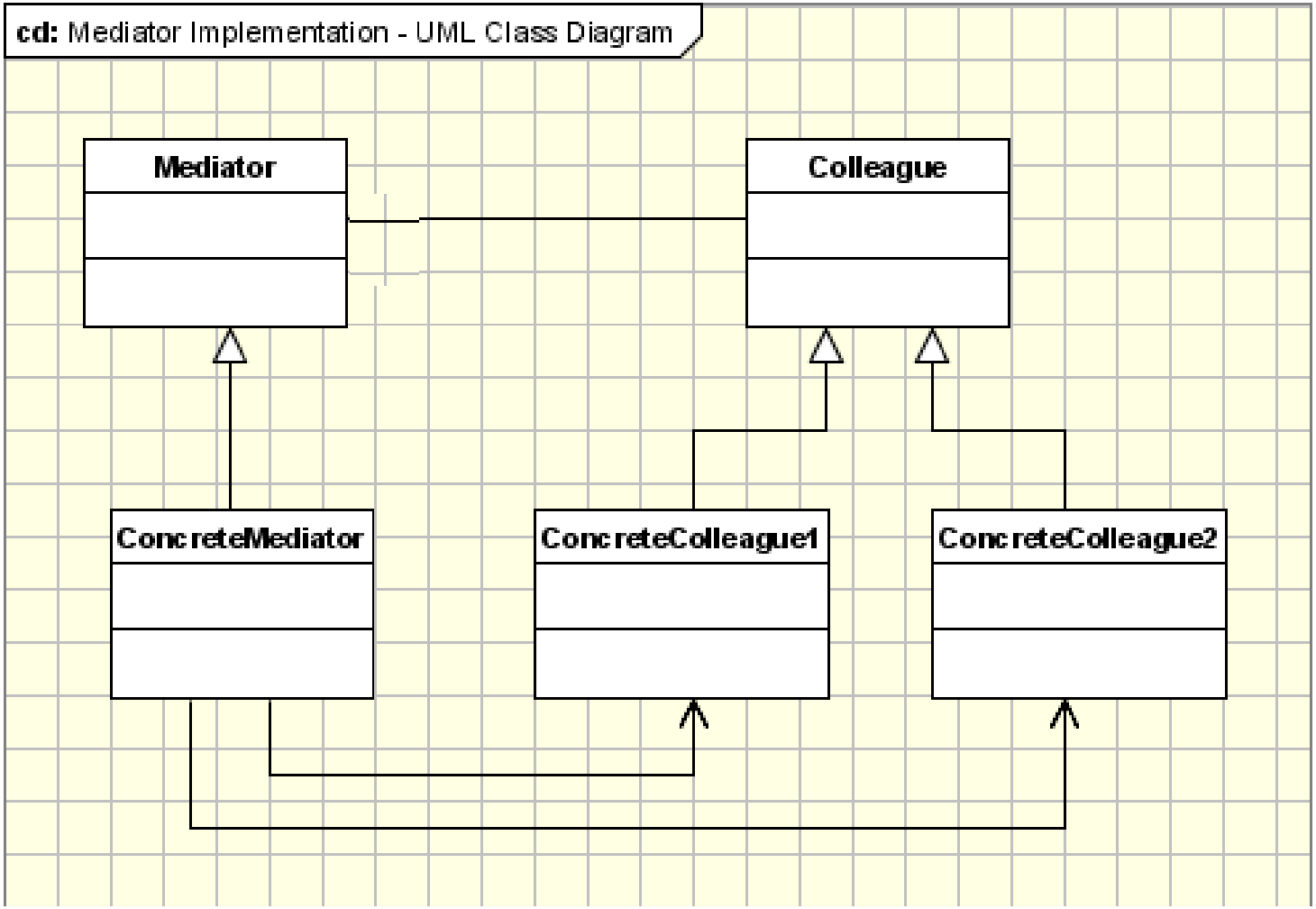
```
List<SimpleJasonAgent> agentList= new
    ArrayList<SimpleJasonAgent> ();
//add elements to agentList

Iterator<SimpleJasonAgent> it = agentList.iterator();
for (; it.hasNext();)
{
    SimpleJasonAgent a = it.next();
    //do something with a
}
```

Mediator

- Define an object that encapsulates how a set of objects interact
- Promotes loose coupling by keeping objects from referring to each other explicitly
- Lets you vary their interaction independently



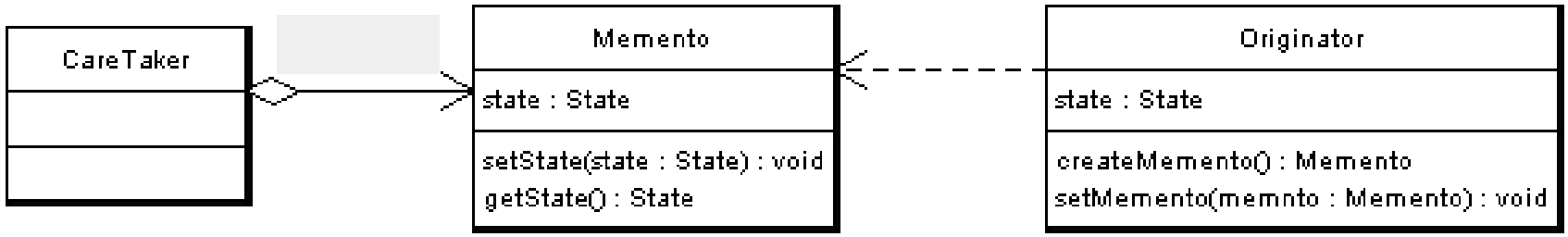
Mediator


cd: Mediator Implementation - UML Class Diagram
Mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Memento

- AKA Token
- Use to restore an object back to one of its previous states
  - e.g. undo function in MS Office
- Capture and externalize an object's internal state so that the object can be returned to this state later
- Encapsulation is not violated

Memento


CareTaker
Memento
state : State
setState(state : State) : void
getState() : State
Originator
state : State
createMemento() : Memento
setMemento(memnto : Memento) : void

Memento

- Originator
  - The object that knows how to save itself : by creating a memento object
  - Makes use of memento object to restore its previous state
- Memento
  - Stores internal state of the Originator object
  - Written and read by the originator, maintained by the caretaker
- Caretaker
  - Responsible for keeping memento objects
  - knows why and when the Originator needs to save and restore itself

Observer

- AKA Dependents, Publish-Subscribe
- Use when there is one to many relationship between objects and if one object is modified, its dependent objects are notified
- The change of a state in one object is reflected in another object without keeping the objects tightly coupled
- The framework can be easily enhanced in future with new observers with minimal changes
- View part of the ‘model-view-controller’
- Extensively used in event management

Observer

- **Observable (Subject)** - interface or abstract class defining the operations for attaching and de-attaching observers to the client

- **ConcreteObservable** - concrete Observable class. Maintains the state of the object and when a change in the state occurs notifies the attached **Observers**

- **Observer** - interface/abstract class defining the operations to be used to notify this object

- **ConcreteObserverA, ConcreteObserver2** - concrete **Observer** implementation

Observer

```
public class ConcreteObservable{
     private List<Observer> observers = new ArrayList<Observer>();
     private int state;
     public int getState() {
               return state;
     }
    public void setState(int state) {
               this.state = state;
               notifyAllObservers();
     }
    public void attach(Observer observer){
               observers.add(observer);
     }
     private void notifyAllObservers(){
               for (Observer observer : observers) {
                         observer.update();
                }
      }
}
```

References

- http://www.oodesign.com/
- http://sourcemaking.com/design_patterns
- http://www.tutorialspoint.com/design_pattern/index.htm