DESIGN PATTERNS

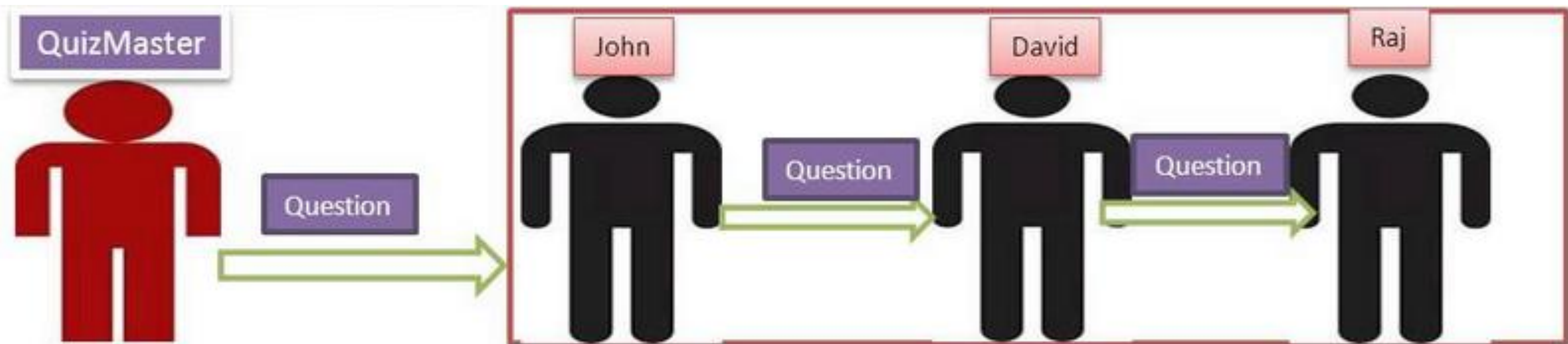
COURSE 6

PREVIOUS COURSE

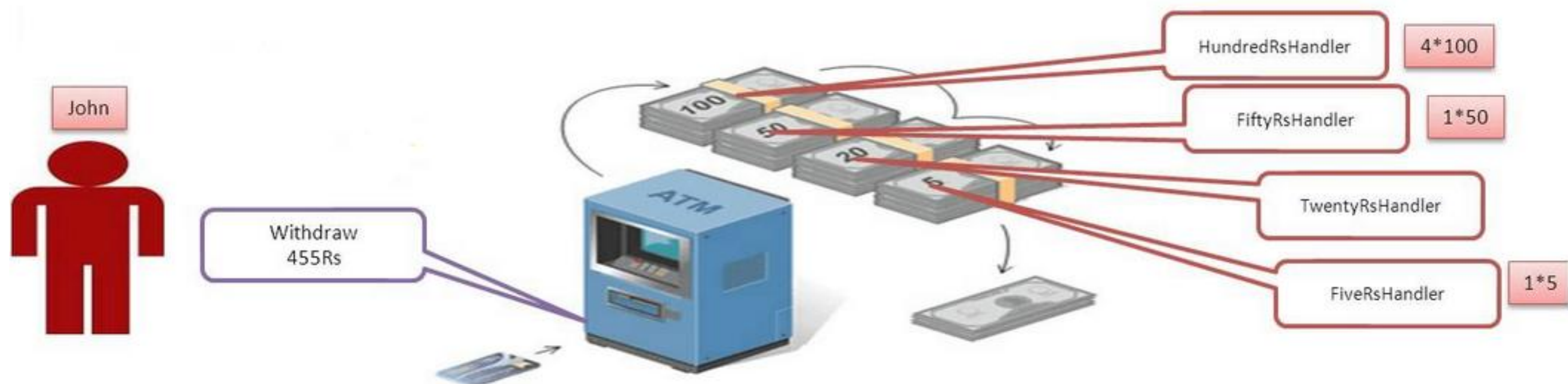
- **Structural patterns**
  - Adapter
  - Bridge
  - Façade
  - Flyweight
  - Proxy
  - Composite
  - Decorator
- **Behavioral patterns**
  - Chain of responsibility

CHAIN OF RESPONSIBILITY

- Only one receiver in the chain handles the Request



- One or more Receivers in the chain handles the Request



BEHAVIORAL PATTERNS

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

BEHAVIORAL PATTERNS

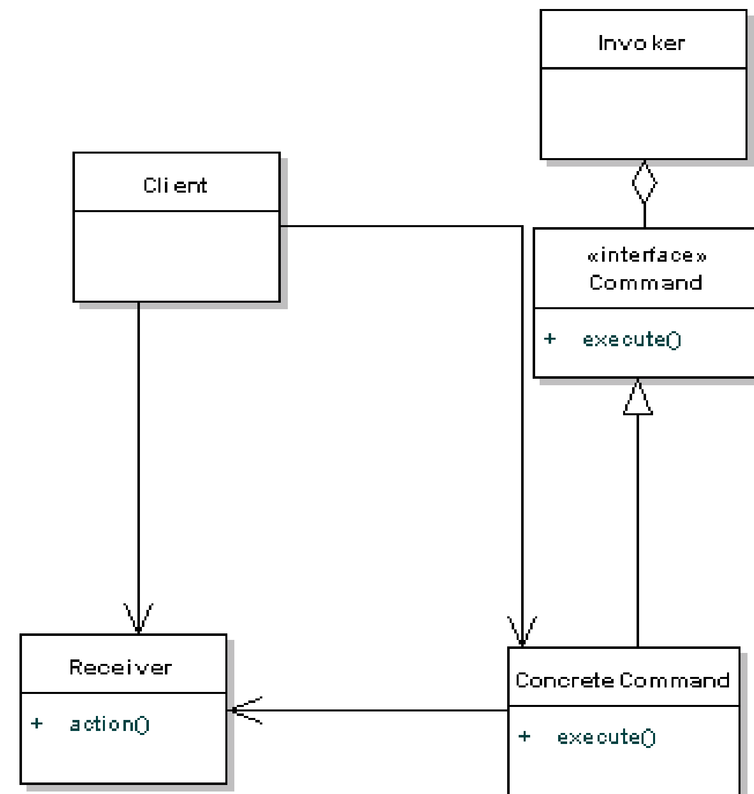
- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

COMMAND

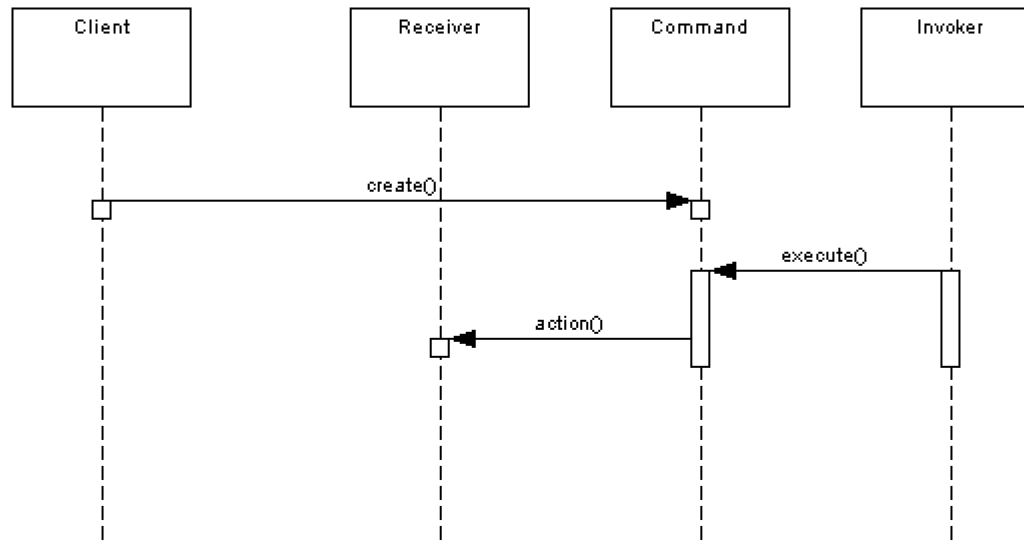
- **Intent**
  - encapsulate a request in an object
  - allows the parameterization of clients with different requests
  - allows saving the requests in a queue
- **Problem**
  - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request

COMMAND

- **Command**
  - declares an interface for executing an operation
- **ConcreteCommand**
  - defines a binding between a Receiver object and an action
  - implements Execute by invoking the corresponding operation(s) on Receiver
- **Client**
  - creates a ConcreteCommand object and sets its receiver
- **Invoker**
  - asks the command to carry out the request
- **Receiver**
  - knows how to perform the operations associated with carrying out the request.



COMMAND



❑The client (main program) creates a concrete Command object and sets its Receiver.

❑The Invoker issues a request by calling execute on the Command object. The concrete Command object invokes operations on its Receiver to carry out the request.

❑The key idea here is that the concrete command registers itself with the Invoker and the Invoker calls it back, executing the command on the Receiver.

COMMAND

- **Example**
  - adding actions to menus in java
  - Create a class that Extends ActionListerner interface and overwrite actionPerformed () method

```
puclic class MyActionHandler extends ActionListener {
  public void actionPerformed(ActionEvent e) {
      Object o = e.getSource();
       if (o = fileNewMenuItem)  doFileNewAction();
       else if (o = fileOpenMenuItem) doFileOpenAction();
       else if (o = fileOpenRecentMenuItem) doFileOpenRecentAction();
       else if (o = fileSaveMenuItem) doFileSaveAction();
      // and more ... }
}

FileOpenMenuItem fomi = new FileOpenMenuItem(“OpenFile”)
fomi.addActionListener(new MyActionHandler());
```

COMMAND

- **Example**
  - adding actions to menus in java
  - If we follow the command pattern first we create a command and after that each menu entry will implement the command

```
public interface Command { public void execute(); }

public class FileOpenMenuItem extends JMenuItem implements
   Command {
     public void execute() { // your business logic goes here }
}

FileOpenMenuItem fomi = new FileOpenMenuItem(“OpenFile”)
fomi.addActionListener(e->{
            Command command = (Command)e.getSource();
            command.execute();
});
```

COMMAND

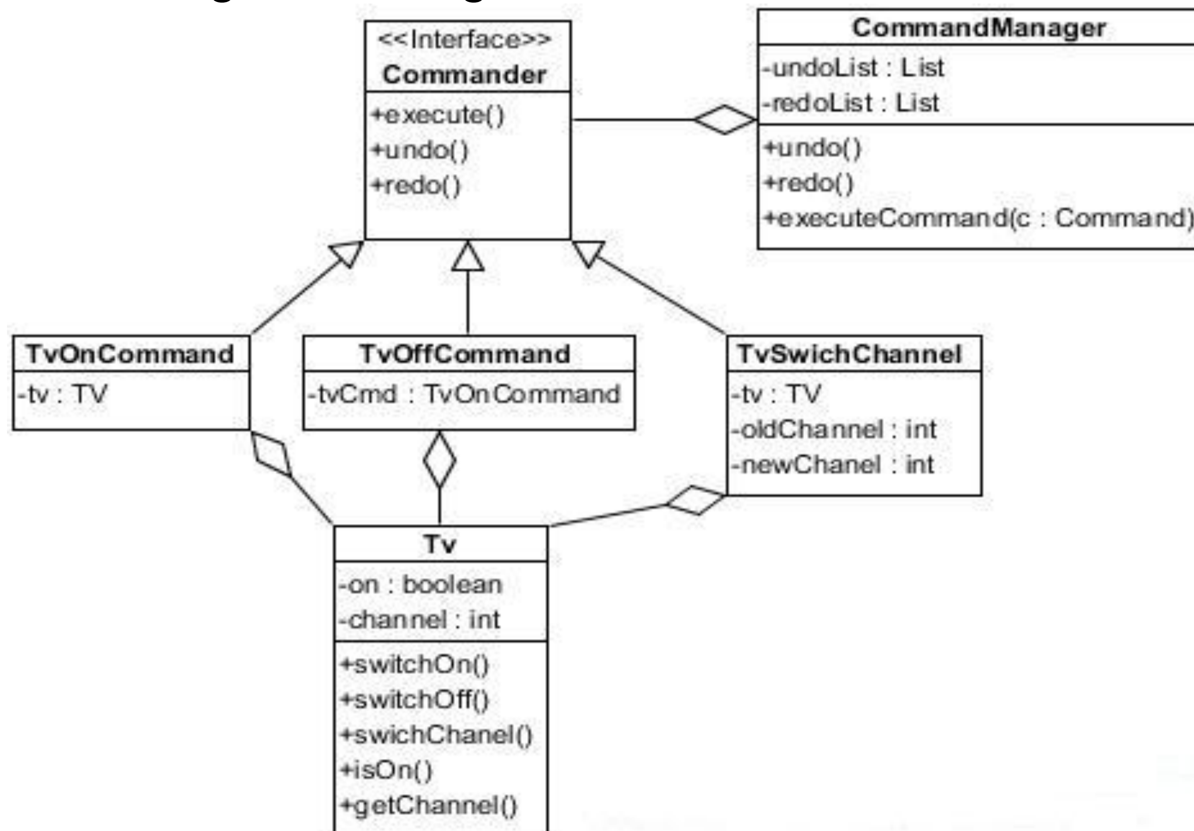
- **Java API examples**
  - ActionListener
  - Comparator
  - Runnable / Thread

COMMAND

- Applicability
  - Parameterizes objects depending on the action they must perform
  - Specifies or adds in a queue and executes requests at different moments in time
  - Offers support for undoable actions (the Execute method can memorize the state and allow going back to that state)
  - Structures the system in high level operations that based on primitive operations
  - Decouples the object that invokes the action from the object that performs the action. Due to this usage it is also known as Producer - Consumer design pattern.

COMMANDER. EXERCISE

- **Implement**
  - Undo/redo operation for a TV remote stating from the following class diagram



COMMAND

- **Advantages**
  - Command decouples the object that invokes the operation from the one that knows how to perform it.
  - Commands are first-class objects. They can be manipulated and extended like any other object.
  - You can assemble commands into a composite command. In general, composite commands are an instance of the Composite pattern.
  - It's easy to add new Commands, because you don't have to change existing classes.

- **Disadvantages**
  - Proliferation of little classes, that are more readable

BEHAVIORAL PATTERNS

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

INTERPRETER

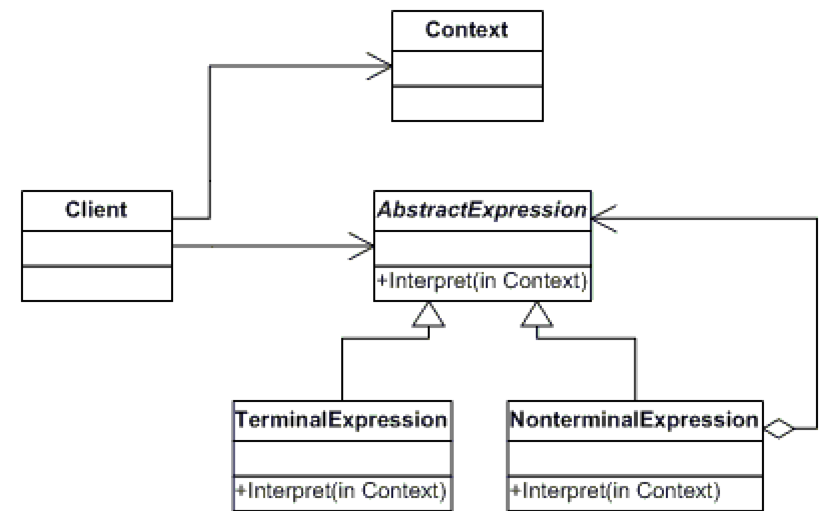
- **Intent**
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
  - Map a domain to a language, the language to a grammar, and the grammar to a hierarchical object-oriented design

- **Problem**
  - A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine"

INTERPRETER. EXAMPLE

- **Language translation**
  - SQL parsing
  - Symbol processing engine
  - Music
    - Grammar = musical notes
    - Interprets = musicians, playing the music

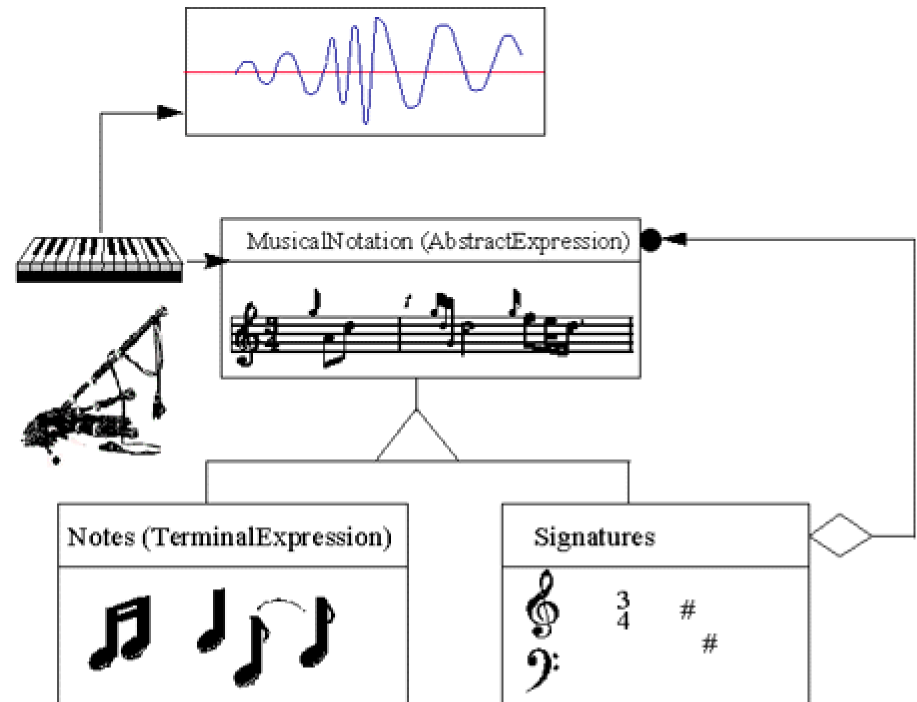
INTERPRETER. STRUCTURE



- **Client**.
  - Client objects build the tree of expressions that represent the commands to be executed, often with the help of a parser class.
  - The Interpret method of the top item in the tree is then called, passing any context object, to execute all of the commands in the tree.
- **Context**.
  - The context class is used to store any information that needs to be available to all of the expression objects.
  - If no global context is required this class is unnecessary.
- **AbstarctExpression**.
  - This abstract class is the base class for all expressions.
  - It defines the Interpret method, which must be implemented for each subclass.
- **TerminalExpression**.
  - Terminal expressions are those that can be interpreted in a single object.
  - These are created as concrete subclasses of the AbstarctExpression class.
- **NonterminalExpression**.
  - Non-terminal expressions are represented using a concrete subclass of AbstractExpression.
  - These expressions are aggregates containing one or more further expressions, each of which may be terminal or non-terminal.
  - When a non-terminal expression class's Interpret method is called, the process of interpretation includes calls to the Interpret method of the expressions it holds.

INTERPRETER. EXAMPLE

- Musicians are examples of Interpreters.
  - The pitch of a sound and its duration can be represented in musical notation on a staff.
  - This notation provides the language of music.
  - Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.



INTERPRETER. EXAMPLE

- **Evaluation of a post-fix (Reverse Polish Notation) of an arithmetic expression**
  - 7 3 – 2 1 + *
  - The result would be?

INTERPRETER. EXAMPLE

- **Evaluation of a post-fix (Reverse Polish Notation) of an arithmetic expression**
  - 7 3 – 2 1 + *
  - The result would be?
    - 12

INTERPRETER. EXAMPLE

```
public interface Expression { public int interpret(); }

public class Add implements Expression{
   private final Expression leftExpression;
   private final Expression rightExpression;
   public Add(Expression leftExpression, Expression rightExpression ){
           this.leftExpression = leftExpression;
           this.rightExpression = rightExpression;
   }
   @Override
    public int interpret() {
        return leftExpression.interpret() + rightExpression.interpret();
    }
}
```

INTERPRETER. EXAMPLE

```
public class Product implements Expression{
    private final Expression leftExpression;
    private final Expression rightExpression;
    public Add(Expression leftExpression, Expression rightExpression ){
            this.leftExpression = leftExpression;
            this.rightExpression = rightExpression;
    }
    @Override
     public int interpret() {
         return leftExpression.interpret() * rightExpression.interpret();
      }
 }

public class Substract implements Expression{
…
}
```

INTERPRETER. EXAMPLE

```
public class Number implements Expression{
  private final int n;
  public Number(int n){ this.n = n; }
  @Override
  public int interpret() { return n; }
}
```

INTERPRETER. EXAMPLE

```
public class ExpressionUtils {
  public static boolean isOperator(String s) {
     if (s.equals("+") || s.equals("-") || s.equals("*"))    return true;
     else   return false;
   }

    public static Expression getOperator(String s, Expression left, Expression right) {
        switch (s) {
          case "+": return new Add(left, right);
          case "-": return new Substract(left, right);
          case "*": return new Product(left, right);
        }
       return null;
   }
}
```

INTERPRETER. EXAMPLE

```
public class TestInterpreterPattern {
  public static void main(String[] args) {
       String tokenString = "7 3 - 2 1 + *";
       Stack<Expression> stack = new Stack<>();
       String[] tokenArray = tokenString.split(" ");
        for (String s : tokenArray) {
            if (ExpressionUtils.isOperator(s)) {
              Expression rightExpression = stack.pop(), leftExpression = stack.pop();
              Expression operator = ExpressionUtils.getOperator(s, leftExpression,rightExpression);
             int result = operator.interpret();
             stack.push(new Number(result));
          } else {
             Expression i = new Number(Integer.parseInt(s));
            stack.push(i);
         }
   }
System.out.println("( "+tokenString+" ): "+stack.pop().interpret());
} }
```

INTERPRETER

- **Interpreter pattern can be used when we can create a syntax tree for a grammar.**

- **Interpreter pattern requires a lot of error checking and a lot of expressions and code to evaluate them, it gets complicated when the grammar becomes more complicated and hence hard to maintain and provide efficiency.**

- **java.util.Pattern and subclasses of java.text.Format are some of the examples of interpreter pattern used in JDK.**

BEHAVIORAL PATTERNS

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

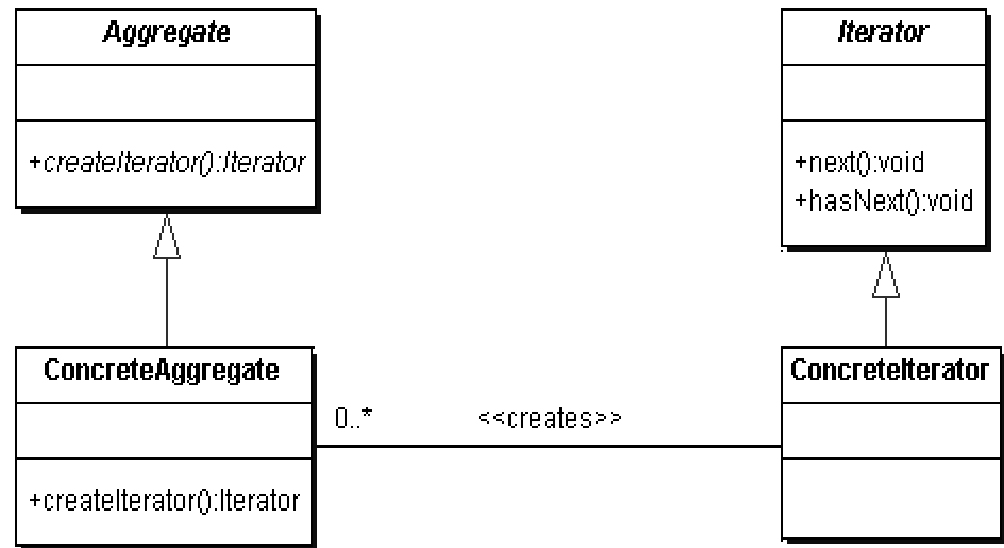
ITERATOR

- **Intent**
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
  - The C++ and Java standard library abstraction that makes it possible to decouple collection classes and algorithms.

- **Problem**
  - An object that provides a standard way to examine all elements of any collection
  - Uniform interface for traversing many different data structures without exposing their implementations
  - Supports concurrent iteration and element removal
  - Removes need to know about internal structure of collection or different methods to access data from different collections

ITERATOR. SUCTURE

- **Aggregate**
  - defines an interface for the creation of the Iterator object.
- **ConcreteAggregate**
  - implements this interface, and returns an instance of the ConcreteIterator.
- **Iterator**
  - defines the interface for access and traversal of the elements
- **ConcreteIterator**
  - implements this interface while keeping track of the current position in the traversal of the Aggregate.



ITERATOR. JDK EXAMPLE

```
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)
}
```

ITERATOR. JDK EXAMPLE

- **All Java collections have a method iterator that returns an iterator for the elements of the collection**
- **Can be used to look through the elements of any kind of collection (an alternative to for loop)**

```
List list = new ArrayList();
... add some elements ...
for (Iterator itr = list.iterator(); itr.hasNext()) {
  BankAccount ba = (BankAccount)itr.next();
  System.out.println(ba);
}
```

ITERATOR. EXAMPLE

- **Iterate through a list of database query records**

```
interface IIterator{
    public boolean hasNext();
    public Object next();
}

interface IContainer{
    public IIterator createIterator();
}
```

ITERATOR. EXAMPLE

```
class RecordCollection implements IContainer{
        private String recordArray[] = {"first","second","third","fourth","fifth"};
        public IIterator createIterator(){
                RecordIterator iterator = new RecordIterator();
                return iterator;
        }
        private class RecordIterator implements IIterator{
                private int index=0;
                public boolean hasNext(){
                        if (index < recordArray.length) return true;
                        else return false;
                }
                public Object next(){
                        if (this.hasNext())   return recordArray[index++];
                        else return null;
                }
        }
}
```

ITERATOR. EXAMPLE

- **Client class**

```
public class TestIterator {
        public static void main(String[] args) {
                RecordCollection recordCollection = new
RecordCollection();
                IIterator iter = recordCollection.createIterator();

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

ITERATOR

- **Consequences**
  - It supports variations in the traversal of an aggregate. Complex aggregates may be traversed in many ways. For example, code generation and semantic checking involve traversing parse trees. Code generation may traverse the parse tree inorder or preorder. Iterators make it easy to change the traversal algorithm: Just replace the iterator instance with a different one. You can also define Iterator subclasses to support new traversals.
  - Iterators simplify the Aggregate interface. Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
  - More than one traversal can be pending on an aggregate. An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

BEHAVIORAL PATTERNS

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

MEDIATOR

- **Intent**
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
  - Design an intermediary to decouple many peers.
  - Promote the many-to-many relationships between interacting peers to "full object status".

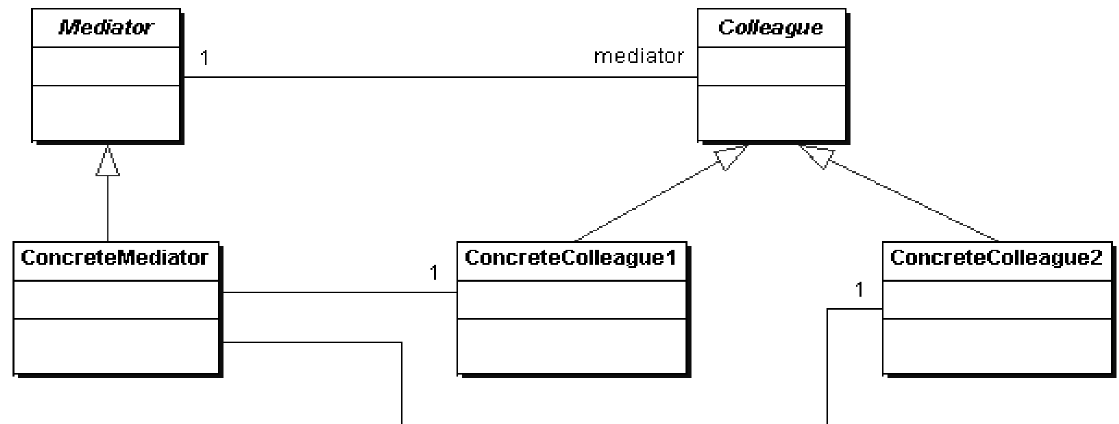
- **Problem**
  - We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

MEDIATOR. EXAMPLES

- **GUI components**
  - Dialog window is a collection of graphic and non-graphic controls
  - Dialog class provides the mechanism to facilitate the interaction between controls
- **JMS (JAVA MESSAGE SERVICE)**
  - Allows applications to subscribe and publish data to other applications
- **Chat application**
  - In a chat application we can have several participants
  - Not a good idea to connect each participant to all the other
  - Solution is to have a hub where all participants will connect
- **Airport control tower**
  - The tower looks after who can take off and land - all communications are done from the airplane to control tower, rather than having plane-to-plane communication.

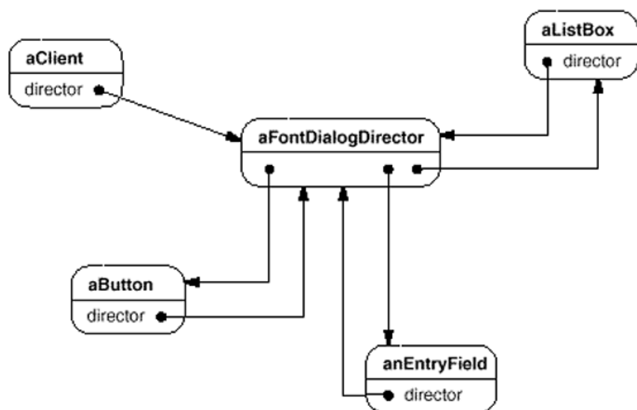
MEDIATOR. STRUCTURE

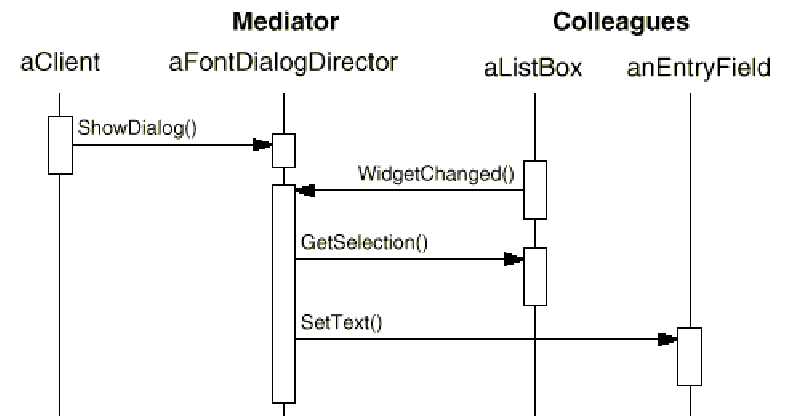
- **Mediator**
  - defines an interface for communicating with Colleague objects
- **ConcreteMediator**
  - knows and maintains its colleagues
  - implements cooperative behavior by coordinating Colleagues
- **Colleague classes**
  - each Colleague class knows its Mediator object
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another

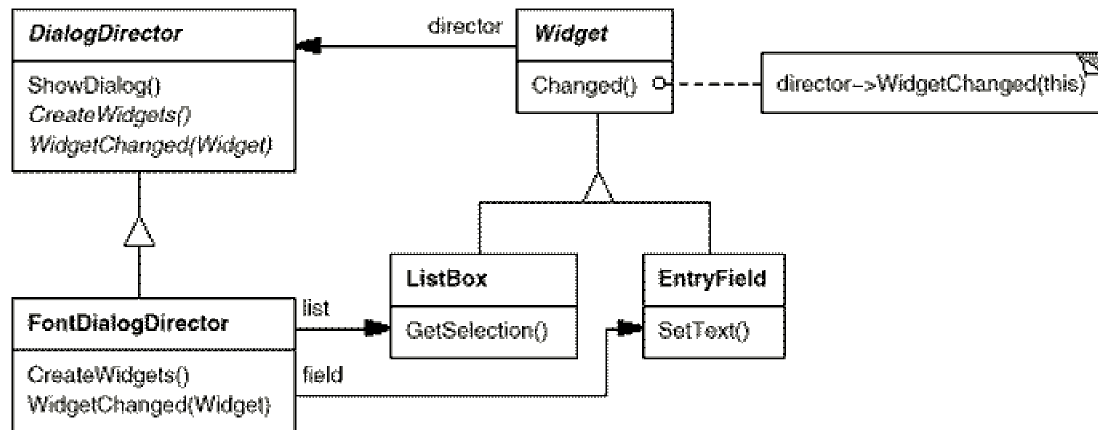


MEDIATOR. EXAMPLE

- GUI interface mediator







MEDIATOR. EXAMPLE

- **Chatroom application**

```
//Mediator interface
public interface Mediator {
  public void send(String message, Colleague colleague);
}

//Colleage interface
public abstract Colleague{
  private Mediator mediator;
  public Colleague(Mediator m) {  mediator = m; }

  //send a message via the mediator
  public void send(String message) { mediator.send(message, this); }

  //get access to the mediator
  public Mediator getMediator() {return mediator;}
  public abstract void receive(String message);
}
```

MEDIATOR. EXAMPLE

```
public class ApplicationMediator implements Mediator {
  private ArrayList<Colleague> colleagues;

  public ApplicationMediator() {   colleagues = new ArrayList<Colleague>();   }

  public void addColleague(Colleague colleague) {   colleagues.add(colleague);  }

  public void send(String message, Colleague originator) {
    //let all other screens know that this screen has changed
    for(Colleague colleague: colleagues) {
      //don't tell ourselves
      if(colleague != originator) {
            colleage.receive(message);
      }
    } } }
```

MEDIATOR. EXAMPLE

```
// concrete colleague
public class ConcreteColleague extends Colleague {
  public void receive(String message) {
    System.out.println("Colleague Received: " + message);
  }
}

// concrete colleague
public class MobileColleague extends Colleague {
  public void receive(String message) {
    System.out.println("Mobile Received: " + message);
  }
}
```

MEDIATOR. EXAMPLE

```
// client class
public class Client {
  public static void main(String[] args) {
    ApplicationMediator mediator = new ApplicationMediator();

    ConcreteColleague desktop = new ConcreteColleague(mediator);
    ConcreteColleague mobile = new MobileColleague(mediator);

    mediator.addColleague(desktop);
    mediator.addColleague(mobile);

    desktop.send("Hello World");
    mobile.send("Hello");
  }
}
```

MEDIATOR

- **When to use mediator pattern?**
  - When one or more objects must interact with several different objects.
  - When centralized control is desired
  - When simple object need to communicate in complex ways.
  - When you want to reuse an object that frequently interacts with other objects

MEDIATOR

- **Benefits**
  - Increases the reusability of the objects supported by the Mediator by decoupling them from the system.
  - Simplfies maintenance of the system by centralizing control logic.
  - Simplifies and reduces the variety of messages sent between objects in the system.
  - Partition a system into pieces or small objects.
  - Centralize control to manipulate participating objects.
  - Most of the complexity involved in managing dependencies is shifted from other objects to the Mediator object.This makes other objects easier to implement and maintain.

- **Disadvantages:**
  - Without proper design, the Mediator object itself can become overly complex.

BEHAVIORAL PATTERNS

- **Chain of responsibility**
  - A way of passing a request between a chain of objects
- **Command**
  - Encapsulate a command request as an object
- **Interpreter**
  - A way to include language elements in a program
- **Iterator**
  - Sequentially access the elements of a collection
- **Mediator**
  - Defines simplified communication between classes
- **Memento**
  - Capture and restore an object's internal state
- **Null Object**
  - Designed to act as a default value of an object
- **Observer**
  - A way of notifying change to a number of classes
- **State**
  - Alter an object's behavior when its state changes
- **Strategy**
  - Encapsulates an algorithm inside a class
- **Template method**
  - Defer the exact steps of an algorithm to a subclass
- **Visitor**
  - Defines a new operation to a class without change

MEMENTO

- **Intent**
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
- **Problem**
  - Need to restore an object back to its previous state

MEMENTO. EXAMPLES

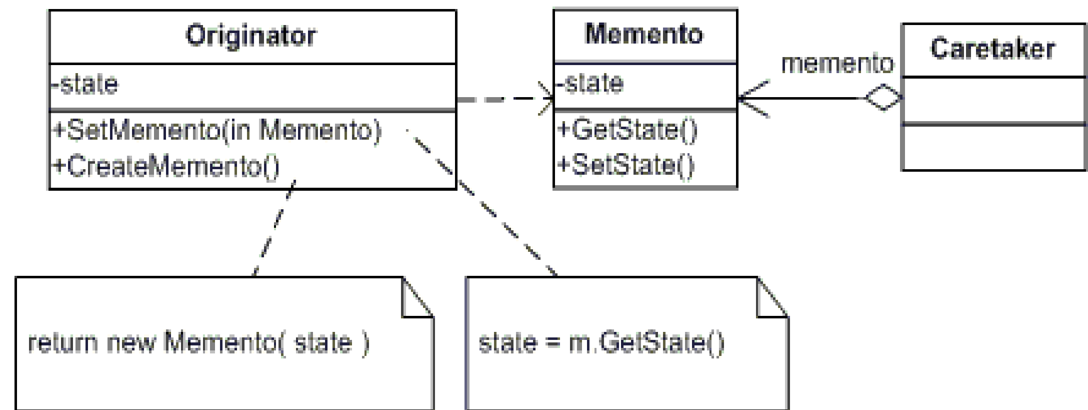
- **Undo and restore operations in most software.**

- **Database transactions**
  - A transaction can contain multiple operations on the database
  - Each operation can succeed or fail
  - A transaction guarantees that if all operations succeed, the transaction would commit and would be final
  - Rolling back mechanism uses the memento design pattern

- **Browser history**

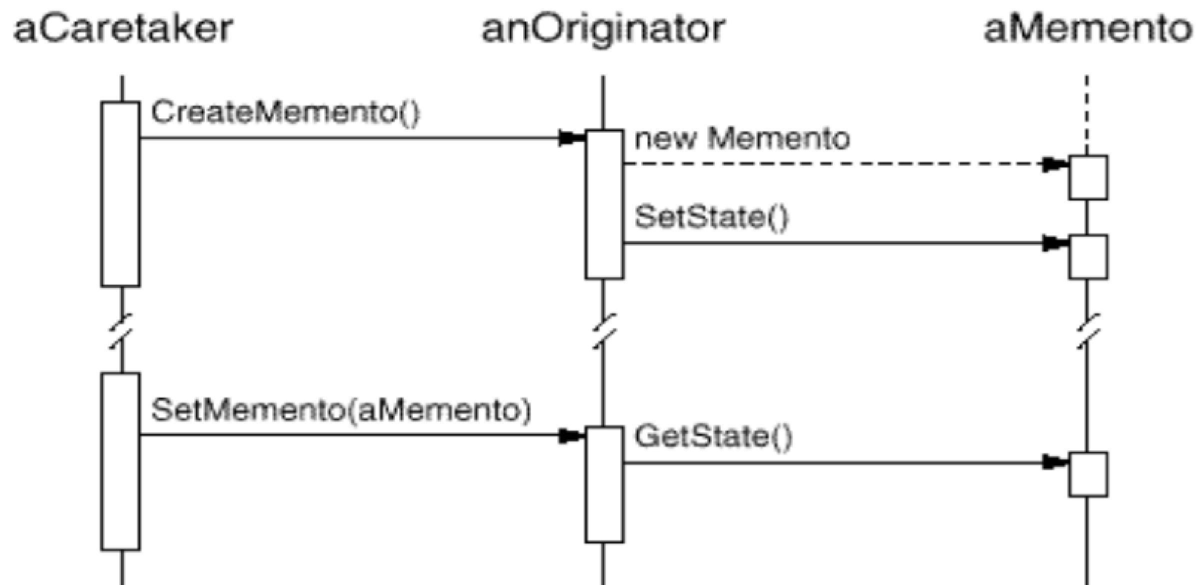
- **Persistency**
  - save / load state between executions of program

MEMENTO. STRUCTURE



- **Originator - the object that knows how to save itself**
  - The state variable contains information that represents the state of the Originator object. This is the variable that will be saved and restored.
  - The CreateMemento method is used to save the state of the Originator.
  - The SetMemento method restores the Originator by accepting a Memento object, unpackage it, and sets its state variable using the state variable from the Memento
- **Caretaker - the object that knows why and when the Originator needs to save and restore itself.**
- **Memento stores the historical information of the Originator. The information is stored in its state variable.**

MEMENTO



- **Organizer**
  - creates a memento containg a snapshot of its current state and uses the memento to restore its internal state
- **Memento**
  - hods internal state of organizer
- **Caretaker**
  - responsable for kipping the memento

MEMENTO

```
class Originator {
    private String state;

    public void set(String state) {
        System.out.println("Originator: Setting state to" + state);
         this.state = state;
     }

    public Object saveToMemento() {
        System.out.println("Originator: Saving to memento");
        return new Memento(state);
   }
//continue on next page
```

MEMENTO

```
public void restoreFromMemento(Object o) {
            if (o instanceof Memento) {
                  Memento m = (Memento) o;
                  state = m.getSavedState();
                  System.out.println("Originator:State after restoring from Memento:" +
state);
             }
     }

private static class Memento {
   private String state;

    public Memento(String stateToSave) { state = stateToSave; }

     public String getSavedState() {  return state;  }
  }
}
```

MEMENTO

```
class CareTaker {
        private List<Object> savedStates = new ArrayList<>();

        public void addMemento(Object m) {
                savedStates.add(m);
        }

        public Object getMemento(int index) {
                return savedStates.get(index);
        }
}
```

MEMENTO

```
public class MementoPatternExample {
        public static void main(String[] args) {
                CareTaker careTaker = new CareTaker();
                Originator originator = new Originator();
                originator.set("State1");
                originator.set("State2");

                careTaker.addMemento(originator.saveToMemento());
                originator.set("State 3");

                careTaker.addMemento(originator.saveToMemento());
                originator.set("State 4");

                originator.restoreFromMemento(careTaker.getMemento(0));
        }
}
```

MEMENTO

- **Benefits**
  - Since object oriented programming dictates that objects should encapsulate their state it would violate this law if objects' internal variables were accessible to external objects. The memento pattern provides a way of recording the internal state of an object in a separate object without violating this law
  - The memento eliminates the need for multiple creation of the same object for the sole purpose of saving its state.
  - The memento simplifies the Originator since the responsibility of managing Memento storage is no longer centralized at the Originator but rather distributed among the Caretakers

- **Drawbacks**
  - The Memento object must provide two types of interfaces: a narrow interface to the Caretaker and a wide interface to the Originator. That is, it must acts like a black box to everything except for the class that created it.
  - Using Mementos might be expensive if the Originator must store a large portion of its state information in the Memento or if the Caretakers constantly request and return the Mementos to the Originator.