Mediator

• Many objects (called the colleagues) have connections between them

• The connections between the objects are managed by a central object: the mediator

Mediator: implementation

• The accesses to the colleagues may be implemented according to the observer pattern or simply by delegation

• The colleagues may pass a reference to themselves (this) within the call to the mediator, to identify the callers
Mediator: an object diagram

State, Interpreter and Visitor Patterns

- These patterns will be used in relation with the analysis of languages (XML or other)
- We will thus study SAX, DOM and Javacc (Java compiler's compiler) in relation with these patterns.
- (See [http://www.epflpress.org/Book_Pages/petitpierre.html](http://www.epflpress.org/Book_Pages/petitpierre.html))
The State Pattern is a behavioral pattern that allows a system to change its behavior when an event occurs. It is used when a class wants to add new behaviors without altering existing code. The pattern uses a State Abstract class to define a common interface for all states, and ConcreteState classes to implement particular states.

The handle() method determines the nature of the event that has occurred. If it is not foreseen in some state, the method takes some corrective action.
State Pattern (one method per action)

- EventGenerator
- Context
  - state
    - action1()
    - action2()
    - action3()
- State
  - action1()
  - action2()
  - action3()
- ConcreteStateA
  - action1()
- ConcreteStateB
  - action1()
  - action3()

The action is determined by the event generator. If it is not foreseen in some state, the default action is called. The later ignores the action or handles the error.

State Pattern (State interface)

```java
abstract class State {
    public void action1(Context context) {}
    public void action2(Context context) {}
    public void action3(Context context) {}
}
```
State Pattern (context)

class Context {
    State state = First.getState();

    public void changeState(State s) {
        state = s;
    }
    public void action1() {
        state.action1(this);
    }
    public void action2() {
        state.action2(this);
    }
    public void action3() {
        state.action3(this);
    }
}

State Pattern (State object)

class Second extends State {
    static State state = null;
    public static State getState() {
        if (state == null) {
            state = new Second();
        }
        return state;
    }
    public void action2(Context context) {
        System.out.println("action 2");
        // go to First State
        context.changeState(First.getState());
    }
    public void action3(Context context) {
        System.out.println("action 3");
        context.changeState(Third.getState());
    }
}
class Context3 {
    enum State {a, b, c};
    State state = State.a;
    public int number;
    public void handle(char inp) {
        switch (state) {
            case a:
                if (inp == '1') {
                    System.out.println("action 1");
                    state = State.b;
                }
                break;
            case b:
                if (inp == '2') {
                    System.out.println("action 2");
                    state = State.a;
                } else if (inp == '3') {
                    System.out.println("action 3");
                    number = 0;
                    state = State.c;
                }
                break;
            ...
        }
    }
}

A SAX Parser with the State Pattern can be used to read XML files.

State Pattern
(based on a switch)
XML: EXtensible Markup Language

<!-- A comment -->

<person>
  <firstName>Robert</firstName>
  <name/>  <!-- empty -->
  <address category="private" duration="permanent">
    <street>Carrigton allee</street>
    <town>Philadelphia</town>
  </address>
</person>

&      '      <      >       "

DTD: Data Type Definition
(defined in the XML standard)

<!ELEMENT person (firstName, name, address)>
<!ELEMENT firstName (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (street, town)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT town (#PCDATA)>
<!ATTLIST address
  category CDATA IMPLIED
  duration (permanent|temporary) "temporary"
>

PCDATA = parsed character data (the &gt; are translated)
Commande ATTLIST

<!--ATTLIST symbolOfTheElement
    symbolOfTheAttribute1 attributeType1 characteristic1
    symbolOfTheAttribute2 attributeType2 characteristic2
 -->
**characteristics**
possibles dans une DTD

"defaultValue“ used after an enumeration, it represents the default value, used if the attribute is not defined

#IMPLIED the attribute is optional

#REQUIRED the attribute is mandatory

---

**Example of XML file**

In order to dump a table of MySQL in XML

C:\>mysqldump --xml databaseName tableName
SAX: Simple API for XML

SAXParser

MyHandler

XML data refers to DTD

DefaultHandler

SAX Handler (available in J2SE)

class MySAXHandler extends DefaultHandler {
    Locator locator = null;
    // ==
    // SAX locator
    // ==
    public void setDocumentLocator(Locator l) {}

    // ==
    // SAX ErrorHandler methods
    // ==
    public void error(SAXParseException e) {}
    public void fatalError(SAXParseException e) throws SAXParseException {}
    // dump warnings too
    public void warning(SAXParseException err) {}

    // ==
    // SAX Handler methods
    // ==
    public void startDocument() {
        System.out.println("Start document");
    }

    public void endDocument() {}
    public void startElement(String uri, String name, String qName, {})
    public void endElement(String uri, String name, String qName, {})
    public void characters(char[] ch, int start, int length) {}
    public void processingInstruction(String target, String data) {}
    public void ignorableWhitespace(char[] ch, int start, int length) {}
Exercise 5.1

Create a program to read the following file:

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE a SYSTEM "test6.dtd">
<a>
  <b>
    <c>xxx xx</c>
    <c>yyyy y</c>
    <c>zz zzz</c>
  </b>
  <d/>
  <b>
    <c>u u u</c>
  </b>
</a>
```

Exercise 5.1 (continuation)

The preceding file correspond to this DTD:

```xml
<!ELEMENT a ( (b | d)* )>
<!ELEMENT b (c)>
<!ELEMENT c (#PCDATA)>
<!ELEMENT d EMPTY>
```

```
0 1 /a
\b 3 \b
\b 2 \d
\c 4 \c
\c 5 \chars
```

```
/a \rightarrow \end
```

```
al \rightarrow \a
```
Using SAX

```java
public static void main(String args[]) throws Exception {
    MySAXHandler handler = new MySAXHandler();
    SAXParserFactory factory = SAXParserFactory.newInstance();
    factory.setValidating(true);
    SAXParser saxParser = factory.newSAXParser();
    File readFile = new File(args[i]);
    try {
        saxParser.parse(readFile, handler);
    } catch (Exception e) {
        System.out.println("Error caught: " + e.getMessage());
    }
}
```

SAX + State Pattern: My handler

```java
/**
 * SAX Handler methods
 */

State state = state0;
public void startElement( String uri,
                           String name,
                           String qName,
                           Attributes atts ) {
    state = state.handle(0, qName);
}
```
SAX + State Pattern: a state

```java
class State0 extends State {
    State handle(int no, String s) {
        if ((s.equals("a")) && (no == 0))
            return state1;
        else
            error(s, no);
        return null;
    }
}
```

SAX + State Pattern

In the previous case, a switch statement to implement the states would be more efficient!
Exercise 5.2

• Use the state pattern to create a tokenizer that decode the following tokens:
  “x” “abc” unknown (all other characters)
• It should be capable of returning the following tokens
  “a” “x” “a” “abc”
  from this sequence
  “axaabc”

Exercise 5.2

A test sequence and an object that contains part of the solution is available (see list of exercises).

// read the token located at the beginning of the buffer,
// from 0 to the current position
public void extractImage()

// Same thing when the first character is not a token
public void extractFirstChar() {

// Called by the client to get the next token
public int getNext() throws IOException {

// Sets the next state
public void setState(State _state) {
Exercise 5.2

For example, for the tokens “abc” “xyz”, the state machine tries the following paths:

State0 ‘a’ (continuation in State1), ‘x’ (continuation in State2), other (extract the first character as the next token and return unknown)

State1 ‘b’ (continuation in State3), other (idem)

State3 ‘c’ (extractImage and return the token number)

State2 ‘y’ (continuation in State4), other (idem)

State4 ‘z’ (extractImage and return the token number)

Exercise 5.2: (JUnit Test)

```java
@Test
public final void test4() throws IOException {
    assertTrue("a, x, a, abc not found",
        readTokens(
            new int[] {0,1,0,2},
            new String[]{"a","x","a","abc"},
            new ByteArrayInputStream("axaabc".getBytes())
        )
    );
}
```
DOM: document object model (tree)

<address category="private" duration="permanent">

DOM: initialization

```java
try {
    DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
    factory.setValidating(true);
    DocumentBuilder builder = factory.newDocumentBuilder();
    builder.setErrorHandler(new ErrorHandler() {
        public void error(SAXParseException exception) {...}
        public void fatalError(SAXParseException exception) {...}
        public void warning(SAXParseException exception) {...}
    });
    document = builder.parse(new File(argv[0]));
    PrintTree.print(document, "\n");
} catch (Exception ioe) {
    ioe.printStackTrace();
}
```
DOM: walking through the tree

```java
public static void print(Node node, String indentation) {
    System.out.print(indentation + node.getNodeValue());
    if (node.hasAttributes()) {
        NamedNodeMap map = node.getAttributes();
        for (int j = 0; j < map.getLength(); j++)
            System.out.println(indentation + map.item(j).getNodeName());
    }
    NodeList nl = node.getChildNodes();
    if (nl != null)
        for (int i = 0; i < nl.getLength(); i++)
            print(nl.item(i), (indentation + "  ")
```
Interpreter for the previous notation
(object diagram)

Interpreter: class diagram