Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Grammar & Classes

Given a language defined by a grammar like:

\[ R ::= R_1 R_2 R_3 \]

you create a class for each rule

The classes can be used to construct a tree that represents elements of the language.
Example - Boolean Expressions

BooleanExpression ::= 
    Variable | 
    Constant | 
    Or | 
    And | 
    Not | 
    BooleanExpression

And ::= '( BooleanExpression 'and' BooleanExpression ')
Or ::= '( BooleanExpression 'or' BooleanExpression ')
Not ::= 'not' BooleanExpression

Constant ::= 'true' | 'false'
Variable ::= String
Sample Expression

\(((\text{true or } x) \text{ or } (w \text{ and } x))\)

Evaluate with
\[
\begin{align*}
x &= \text{true} \\
w &= \text{false}
\end{align*}
\]
Sample Classes

public interface BooleanExpression{
    public boolean evaluate( Context values );
    public String toString();
}

public class And implements BooleanExpression {
    private BooleanExpression leftOperand;
    private BooleanExpression rightOperand;

    public And( BooleanExpression leftOperand, BooleanExpression rightOperand) {
        this.leftOperand = leftOperand;
        this.rightOperand = rightOperand;
    }

    public boolean evaluate( Context values ) {
        return leftOperand.evaluate( values ) && rightOperand.evaluate( values );
    }

    public String toString(){
        return "(" + leftOperand.toString() + " and " + rightOperand.toString() + ")";
    }
}
public class Constant implements BooleanExpression {
    private boolean value;
    private static Constant True = new Constant(true);
    private static Constant False = new Constant(false);

    public static Constant getTrue() { return True; }

    public static Constant getFalse() { return False; }

    private Constant(boolean value) { this.value = value; }

    public boolean evaluate(Context values) { return value; }

    public String toString() {
        return String.valueOf(value);
    }
}
public class Variable implements BooleanExpression {

    private String name;

    private Variable( String name ) {
        this.name = name;
    }

    public boolean evaluate( Context values ) {
        return values.getValue( name );
    }

    public String toString() { return name; }
}

public class Context {
    Hashtable<String,Boolean> values = new Hashtable<String,Boolean>();

    public boolean getValue( String variableName ) {
        return values.get( variableName );
    }

    public void setValue( String variableName, boolean value ) {
        values.put( variableName, value );
    }
}
((true or x) or (w and x))

```java
public class Test {
    public static void main( String args[] ) throws Exception {
        BooleanExpression left =
            new Or( Constant.getTrue(), new Variable( "x" ) );
        BooleanExpression right =
            new And( new Variable( "w" ), new Variable( "x" ) );

        BooleanExpression all = new Or( left, right );

        System.out.println( all );
        Context values = new Context();
        values.setValue( "x", true );
        values.setValue( "w", false );

        System.out.println( all.evaluate( values ) );
    }
}
```
Consequences

It's easy to change and extend the grammar

Implementing the grammar is easy

Complex grammars are hard to maintain

    Use JavaCC or SmaCC instead

Adding new ways to interpret expressions

    The visitor pattern is useful here

Complicates design when a language is simple

Supports combinations of elements better than implicit language
Implementation

The pattern does not talk about parsing!

Flyweight

If terminal symbols are repeated many times using the Flyweight pattern can reduce space usage.

Composite
Abstract syntax tree is an instance of the composite

Iterator
Can be used to traverse the structure

Visitor
Can be used to place behavior in one class
Observer

One-to-many dependency between objects

When one object changes state,
    all its dependents are notified and updated automatically
Structure

**observer A**

**observer B**

GetState()  Update()

**subject**

SetState()  Notify()

**ConcreteSubject**

GetState()  Update()

subjectState

**ConcreteObserver**

subject  observers

Update()

**Observer**

Update()
public class Subject {
    Window display;
    public void someMethod() {
        this.modifyMyStateSomeHow();
        display.addText( this.text() );
    }
}

public class Subject {
    ArrayList observers = new ArrayList();
    public void someMethod() {
        this.modifyMyStateSomeHow();
        changed();
    }

    private void changed() {
        Iterator needsUpdate = observers.iterator();
        while (needsUpdate.hasNext())
            needsUpdate.next().update( this );
    }
}

public class SampleWindow {
    public void update(Object subject) {
        text = ((Subject) subject).getText();
        Thread.sleep(10000).
    }
}

Abstract coupling - Subject & Observer

Broadcast communication

Updates can take too long
## Some Language Support

<table>
<thead>
<tr>
<th>Smalltalk</th>
<th>Java</th>
<th>Ruby</th>
<th>Observer Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
<td>Observer</td>
<td></td>
<td>Abstract Observer class</td>
</tr>
<tr>
<td>Object &amp; Model</td>
<td>Observable</td>
<td>Observable</td>
<td>Subject class</td>
</tr>
</tbody>
</table>

### Smalltalk Implementation

Object implements methods for both Observer and Subject.

Actual Subjects should subclass Model
Java's Observer

Class java.util.Observable

- `void addObserver(Observer o)`
- `void clearChanged()`
- `int countObservers()`
- `void deleteObserver(Observer o)`
- `void deleteObservers()`
- `boolean hasChanged()`
- `void notifyObservers()`
- `void notifyObservers(Object arg)`
- `void setChanged()`

Observable object may have any number of Observers

Whenever the Observable instance changes, it notifies all of its observers

Notification is done by calling the update() method on all observers.

**Interface java.util.Observer**

Allows all classes to be observable by instances of class Observer

---

<table>
<thead>
<tr>
<th>Java</th>
<th>Observer Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Observer</td>
<td>Abstract Observer class</td>
</tr>
<tr>
<td>Observable class</td>
<td>Subject class</td>
</tr>
</tbody>
</table>
class Counter extends Observable {
    public static final String INCREASE = "increase";
    public static final String DECREASE = "decrease";
    private int count = 0;
    private String label;

    public Counter( String label ) { this.label = label; }

    public String label() { return label; }
    public int value() { return count; }
    public String toString() { return String.valueOf( count );}

    public void increase() {
        count++;
        setChanged();
        notifyObservers( INCREASE );
    }

    public void decrease() {
        count--;
        setChanged();
        notifyObservers( DECREASE );
    }
}
Java Observer

class IncreaseDetector implements Observer {
    public void update( java.util.Observable whatChanged,
            java.lang.Object message) {
        if ( message.equals( Counter.INCREASE) ) {
            Counter increased = (Counter) whatChanged;
            System.out.println( increased.label() + " changed to " +
                                increased.value());
        }
    }
}

public static void main(String[] args) {
    Counter test = new Counter();
    IncreaseDetector adding = new IncreaseDetector();
    test.addObserver(adding);
    test.increase();
}
Ruby Example

require 'observer'

class Counter
  include Observable

  attr_reader :count

  def initialize
    @count = 0
  end

  def increase
    @count += 1
    changed
    notify_observers(:INCREASE)
  end

  def decrease
    @count -= 1
    changed
    notify_observers(:DECREASE)
  end
end

class IncreaseDetector
  def update(type)
    if type == :INCREASE
      puts('Increase')
    end
  end
end

count = Counter.new()
puts count.count
count.add_observer(IncreaseDetector.new)
count.increase
count.increase
puts count.count
Implementation Issues
Mapping subjects(Observables) to observers

Use list in subject
Use hash table

```java
public class Observable {
    private boolean changed = false;
    private Vector obs;

    public Observable() {
        obs = new Vector();
    }

    public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }

    public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) {
            obs.addElement(o);
        }
    }
}
```
Observing more than one subject

If an observer has more than one subject how does it know which one changed?

Pass information in the update method
Deleting Subjects

In C++ the subject may no longer exist

Java/Smalltalk observer may prevent subject from garbage collection
Who Triggers the update?

Have methods that change the state trigger update

class Counter extends Observable { // some code removed
    public void increase() {
        count++;  
        setChanged();
        notifyObservers(INCREASE);
    }
}

Have clients call Notify at the right time

class Counter extends Observable { // some code removed
    public void increase() { count++;
    }
}

Counter pageHits = new Counter();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.increase();
pageHits.notifyObservers();
Subject is self-consistent before Notification

class ComplexObservable extends Observable {
    Widget frontPart = new Widget();
    Gadget internalPart = new Gadget();

    public void trickyChange() {
        frontPart.widgetChange();
        internalPart.anotherChange();
        setChanged();
        notifyObservers();
    }
}

class MySubclass extends ComplexObservable {
    Gear backEnd = new Gear();

    public void trickyChange() {
        super.trickyChange();
        backEnd.yetAnotherChange();
        setChanged();
        notifyObservers();
    }
}
Adding information about the change

push models - add parameters in the update method

```java
class IncreaseDetector extends Counter implements Observer { // stuff not shown
    public void update( Observable whatChanged, Object message) {
        if ( message.equals( INCREASE) )
            increase();
    }
}

class Counter extends Observable { // some code removed
    public void increase() {
        count++;
        setChanged();
        notifyObservers( INCREASE );
    }
}
```
Adding information about the change

pull model - observer asks Subject what happened

class IncreaseDetector extends Counter implements Observer {
    public void update( Observable whatChanged ) {
        if ( whatChanged.didUserIncrease() )
            increase();
    }
}

class Counter extends Observable {
    public void increase() {
        count++;
        setChanged();
        notifyObservers();
    }
}


Scaling the Pattern
Java Event Model

AWT/Swing components broadcast events to Listeners

JDK1.0 AWT components broadcast an event to all its listeners

A listener normally not interested all events

Broadcasting to all listeners was too slow with many listeners
Java 1.1+ Event Model

Each component supports different types of events:

Component supports
  ComponentEvent         FocusEvent
  KeyEvent              MouseEvent

Each event type supports one or more listener types:

  MouseEvent
    MouseListener       MouseMotionListener

Each listener interface replaces update with multiple methods

  MouseListener
    mouseClicked()    mouseEntered()
    mousePressed()     mouseReleased()

Listeners
  Only register for events of interest
  Don't need case statements to determine what happened
Small Models

Often an object has a number of fields (aspects) of interest to observers

Rather than make the object a subject make the individual fields subjects
  Simplifies the main object
  Observers can register for only the data they are interested in

VisualWorks ValueHolder

Subject for one value

ValueHolder allows you to:

  Set/get the value
    Setting the value notifies the observers of the change

  Add/Remove dependents
Reactive Programming

datatypes that represent a value 'over time'

Spreadsheets
Elm
Meteor.js

main = lift asText Mouse.position

(144,49)
State
State Pattern

Allow an object to alter its behavior when its internal state changes

The object will appear to change its class
Structure

```
Context
  state ->handle()
  request()

State
  handle()

ConcreteStateA
  handle()

ConcreteStateB
  handle()
```
Oracle seer = new Truthful();
seer.willThereBeAFeeIncreaseNextYear();
seer = new Lying();
seer.willThereBeAFeeIncreaseNextYear();
public class Oracle {
    private final TRUTH = "truth";
    private final LIE = "lie";
    private final RANDOM = "random";

    String state = TRUTH;

    public boolean willThereBeAFeeIncreaseNextYear() {
        if (state == TRUTH)
            blah
        else if (state == LIE)
            more blah
        else if (state == RANDOM)
            random blah
    }
}
class Oracle {
    private State mode = set mode;

    public boolean willThereBeAFeeIncreaseNextYear() {
        return mode.willThereBeAFeeIncreaseNextYear();
    }
}
Example: SDChat Server

Commands

"available"
"login"
"register"
"nickname"
"startconversation"
"quit"
"waitinglist"
"acceptconversation"
"message"
"rejectconnection"
"endconversation"
Server States
public class SDChatServer {

    String handleNickname(String data) {
        if (state != START)
            return someErrorMessage();
        handle the main case
    }

    String handleLogin(String data) {
        if (state != START)
            return someErrorMessage();
        handle main case
    }

    String handleWaitinglist(String data) {
        if (state != AUTHENTICATED)
            return someErrorMessage();
        handle main case
    }

}
class Context {
    private AbstractState state = new StartState();

    public Bar foo(int x) {
        int result = state.foo(x);
        if (someConditionHolds() )
            state = nextState();
        return result;
    }
}
Who defines state Transitions - States

class Context {
    private AbstractState state = new StartState();

    public void foo(int x) {
        state = state.foo(x);
    }

    What if foo returns a value?
Who defines state Transitions - States

class Context {
    private AbstractState state = new StartState();

    public int foo(int x) {
        return state.foo(x, this);
    }

    protected void setState(AbstractState newState) {
        state = newState;
    }
}
Sharing State Objects

Stateless state
  State objects without fields
  Can be shared by multiple contexts

Can store date in context and pass as arguments

Large number of state transitions can be expensive

Only create state once & reuse same object
class Truthful extends Oracle {

    public boolean foo(int x) {
        int result = state.foo(x);
        this.changeClassTo(Random);
        return result;
    }
}
State Verses Strategy

Rate of Change

**Strategy**
Context usually contains just one strategy object

**State**
Context often changes state objects
State Verses Strategy

Exposure of Change

Strategy
Strategies all do the same thing

Client do not see change in behavior of Context

State
States act differently

Client see the change in behavior