CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2015
Doc 12 Interpreter, 4 rules, Coupling
March 12, 2015

Copyright ©, All rights reserved. 2015 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.
Interpreter
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

```
+-----------------+      +-----------------+      +-----------------+
|     Context     |      |  AbstractExpression  |      |  NonTerminalExpression  |
|                 |      | interpret( Context ) |      | abstractExpressions    |
|                 |      |                    |      | interpret( Context )   |
+-----------------+      +-----------------+      +-----------------+
```

```
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

\[((true \lor x) \lor (w \land x))\]

Evaluate with
\[x = \text{true}\]
\[w = \text{false}\]
Sample Classes

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

Thursday, March 12, 15
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))

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
Kent Beck’s 4 Rules of Simple Design
References

Martin Fowler


J. B. Rains

http://www.jbrains.ca/permalink/the-four-elements-of-simple-design

http://blog.thecodewhisperer.com/2013/12/07/putting-an-age-old-battle-to-rest/
The Rules

In order of importance

Passes the tests

Reveals intention (maximizes clarity)

Minimizes duplication

Minimal methods, classes, modules
Technical Spikes

How do you write tests when you don’t know what you are doing?
Coupling
In the Beginning

Parnas (72) KWIC (Simple key word in context) experiment

Read lines of words
Output all circular shifts of all lines in alphabetical order
Circular shift
  remove first word of line and add it to the end of the line
KWIC Solutions

Solution 1
Each major step in processing is a module
Create flowchart and make each major part a module

Solution 2
Modules based on design decisions
List design decisions that are
Difficult
Likely to change
Each module should hide a design decision

Solution 1
More complex
Harder to understand
Much harder to modify
Metrics for Quality

Coupling

Strength of interaction between objects in system

Cohesion

Degree to which the tasks performed by a single module are functionally related
Coupling

Measure of the interdependence among modules

"Unnecessary object coupling needlessly decreases the reusability of the coupled objects"

"Unnecessary object coupling also increases the chances of system corruption when changes are made to one or more of the coupled objects"

Design Goal

The interaction or other interrelationship between any two components at the same level of abstraction within the system be as weak as possible
Types of Modular Coupling
In order of desirability

Data Coupling   (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling   (strongest – least desirable)

Content Coupling   (Unrated)
Data Coupling

Output from one module is the input to another
Using parameter lists to pass items between routines

Common Object Occurrence

Object A passes object X to object B
Object X and B are coupled
A change to X's interface may require a change to B

Example

class ObjectBClass{
    public void message( ObjectXClass X ){
        // code goes here
        X.doSomethingForMe( Object data );
        // more code
    }
}

Data Coupling

Problem

Object A passes object X to object B
X is a compound object
Object B must extract component object Y out of X

B, X, internal representation of X, and Y are coupled

```java
public class HiddenCoupling {
    public bar someMethod(SomeType x) {
        AnotherType y = x.getY();
        y.foo();
        blah;
    }
}
```
Example – Sorting

How to write a general purpose sort
Sort the same list by
   ID
   Name
   Grade

class StudentRecord {
   Name lastName;
   Name firstName;
   long ID;

   public Name getLastName() { return
      lastName; }
   // etc.
}

SortedList cs635 = new SortedList();
StudentRecord newStudent;
//etc.
cs535.add ( newStudent );
class SortedList
{
    Object[] sortedElements = new Object[ properSize ];

    public void add( StudentRecord X )
    {
        // coded not shown
        Name a = X.getLastName();
        Name b = sortedElements[ K ].getLastName();
        if ( a.lessThan( b ) )
            // do something
        else
            // do something else
    }
}
class SortedList{
    Object[] sortedElements = new Object[ properSize ];

    public void add( StudentRecord X ) {
        // coded not shown
        if ( X.lessthan( sortedElements[ K ] ) )
            // do something
        else
            // do something else
    }
}

class StudentRecord{
    private Name lastName;
    private long ID;

    public boolean lessThan( Object compareMe ) {
        return lastName.lessThan( compareMe.lastName );
    }
    etc.
}
interface Comparable {
    public boolean lessThan( Object compareMe );
    public boolean greaterThan( Object compareMe );
    public boolean equal( Object compareMe );
}

class StudentRecord implements Comparable {
    blah
    public boolean lessThan( Object compareMe ) {
        return lastName.lessThan( ((Name)compareMe).lastName );
    }
}

class SortedList {
    Object[] sortedElements = new Object[ properSize ];

    public void add( Comparable X ) {
        // coded not shown
        if ( X.lessThan( sortedElements[ K ] )
            // do something
        else
            // do something else
    }
}
interface Comparing {
    public boolean lessThan( Object a, Object b );
    public boolean greaterThan( Object a, Object b );
    public boolean equal( Object a, Object b );
}

class SortedList {
    Object[] sortedElements = new Object[ properSize ];
    Comparing comparer;
    public SortedList(Comparing y) {comparer = y;}

    public void add( Object X ) {
        // coded not shown
        if ( comparer.lessThan( sortedElements[ K ], X )
            // do something
        else
            // do something else
    }
}
class ByName implements Comparing {
   public boolean lessThan( Object a, Object b ) {
      return ((Student) a).lastName() < ((Student) b).lastName();
   }
   etc.
}

class ByID implements Comparing {
   public boolean lessThan( Object a, Object b ) {
      return ((Student) a).id() < ((Student) b).id();
   }
   etc.
}

SortedList byName = new SortedList( new ByName() );

SortedList byID = new SortedList( new ById());
interface Comparator<T> { int compare(T o1, T o2) }

class SortedList<T> {
   T[] sortedElements = new T[properSize];
   Comparator<T> comparer;
   public SortedList(Comparator<T> y) {comparer = y;}

   public void add( Object X ) {
      // coded not shown
      if ( (comparer.compare( sortedElements[ K ], X ) < 0 )
         // do something
      else
         // do something else
      }
   }
}
Java 8 Solution

```java
SortedList byName = new SortedList( (a, b) -> a.lastName() < b.name());

SortedList byID = new SortedList( (a, b) -> a.id() < b.id());
```
Functor Pattern

Functors are functions that behave like objects

They serve the role of a function, but can be created, passed as parameters, and manipulated like objects

A functor is a class with a single member function

Note 1: Functors violate the idea that a class is an abstraction with operations and state. Beginners should avoid using the Functor pattern, as they can lead to bad habits. The functor pattern is used here only as a last resort.

Note 2: The Command pattern is similar to the Functor pattern, but contains operations and state.
Types of Coupling

Data Coupling (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling (strongest – least desirable)

Content Coupling (Unrated)
Control Coupling

Passing control flags between modules so that one module controls the sequencing of the processing steps in another module

Common Object Occurrence

A sends a message to B
B uses a parameter of the message to decide what to do

class Lamp {
    public static final ON = 0;

    public void setLamp( int setting ) {
        if ( setting == ON )
            // turn light on
        else if ( setting == 1 )
            // turn light off
        else if ( setting == 2 )
            // blink
    }
}

Lamp reading = new Lamp();
reading.setLamp( Lamp.ON );
reading.setLamp( 2 );
Cure

Decompose the operation into multiple primitive operations

class Lamp {
    public void on() {//turn light on }
    public void off() {//turn light off }
    public void blink() {//blink }
}

Lamp reading = new Lamp();
reading.on();
reading.blink();
Is this Control Coupling

class BankAccount {
    public void withdrawal(Float amount) {
        balance = balance - amount;
    }
}

etc.

Is this Control Coupling

class BankAccount {
    public void withdrawal(Float amount) {
        if (balance < amount)
            this.bounceThisCheck();
        else
            balance = balance - amount;
    }
}

etc.
What if the Lamp had 50 settings?
Control Coupling

Common Object Occurrence

A sends a message to B
B returns control information to A

Example: Returning error codes

class Test {
    public int printFile( File toPrint ) {
        if ( toPrint is corrupted )
            return CORRUPTFLAG;
        blah blah blah
    }
}

Test when = new Test();
int result = when.printFile( popQuiz );
if ( result == CORRUPTFLAG )
    blah
else if ( result == -243 )
Cure – Use Exceptions

How does this reduce coupling?

class Test {
    public int printFile( File toPrint ) throws PrintException {
        if ( toPrint is corrupted )
            throws new PrintException();
        blah blah blah
    }
}

try {
    Test when = new Test();
    when.printFile( popQuiz );
}

    
catch ( PrintException printError ) {
        do something
    }
}
Types of Coupling

Data Coupling  (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling  (strongest – least desirable)

Content Coupling  (Unrated)
Global Data Coupling

Global Data is evil
Global Data Coupling

What are the following?

System.out
Integer.MAX_VALUE
Types of Global Data Coupling in increasing order of "badness"

Make a reference to a specific external object

Make a reference to a specific external object, and to methods in the external object

A component of an object-oriented system has a public interface which consists of items whose values remain constant throughout execution, and whose underlying structures/implementations are hidden

A component of an object-oriented system has a public interface which consists of items whose values remain constant throughout execution, and whose underlying structures/implementations are not hidden

A component of an object-oriented system has a public interface which consists of items whose values do not remain constant throughout execution, and whose underlying structures/implementations are hidden

A component of an object-oriented system has a public interface which consists of items whose values do not remain constant throughout execution, and whose underlying structures/implementations are not hidden
Types of Coupling

Data Coupling  (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling  (strongest – least desirable)

Content Coupling   (Unrated)
Internal Data Coupling

One module directly modifies local data of another module

Common Object Occurrences

C++ Friends
Smalltalk reflection
Java reflection
Internal Data Coupling

Implement a debugger without using internal data coupling
Types of Coupling

Data Coupling  (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling  (strongest – least desirable)

**Content Coupling**  (Unrated)
Lexical Content Coupling

Some or all of the contents of one module are included in the contents of another

Common Object Occurrence

C/C++ header files

Decrease coupling by
  Restrict what goes in header file
  C++ header files should contain only class interface specifications