# CS 635 Advanced Object-Oriented Design & Programming
## Spring Semester, 2002
## Doc 14 Interpreter & Factory Method
## Contents



## Reference

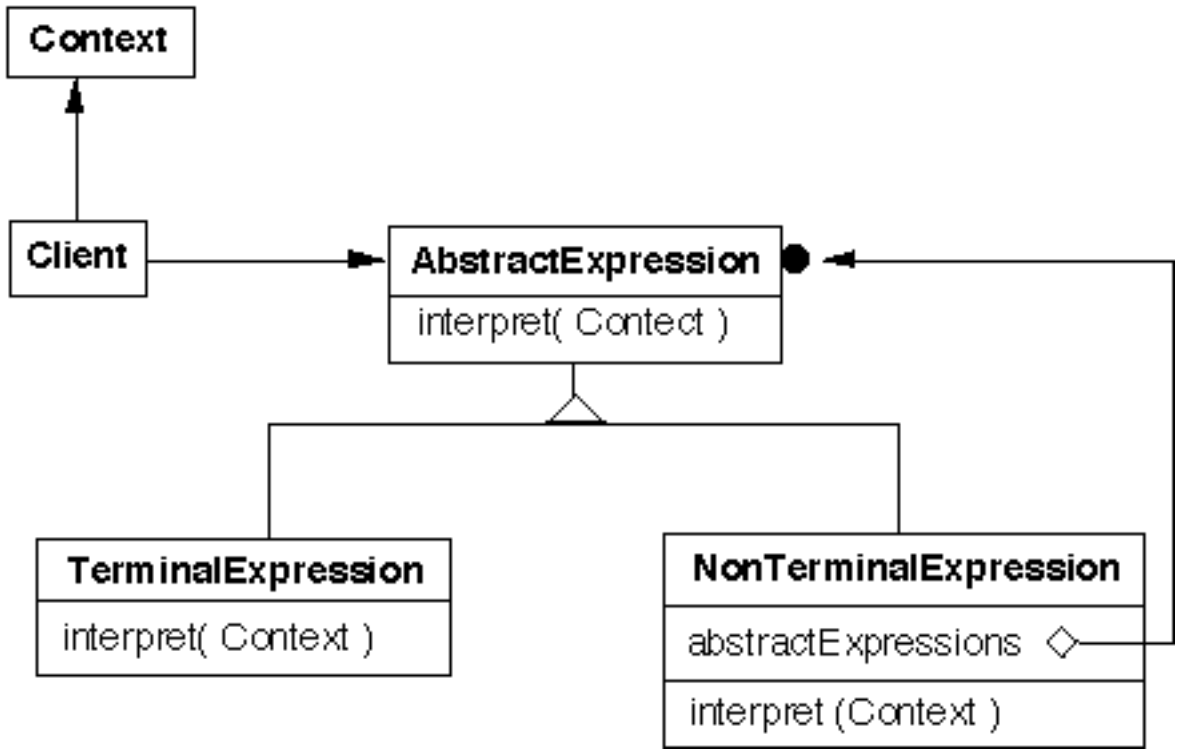
Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, 1995, pp. 107-116, 243-256

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, 1998, pp. 63-89

# Interpreter

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

## Structure




Given a language defined by a simple grammar with rules like:

R ::= R1 R2 ... Rn

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

public interface BooleanExpression{
   public boolean evaluate( Context values );
   public BooleanExpression replace( String varName,
                        BooleanExpression replacement );
   public Object clone();
   public String toString();
}
```

## Sample Use

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

    BooleanExpression all = new And(  left, right );

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

    System.out.println( all.evaluate( values ) );
    System.out.println( all.replace( "x", right ) );
    }
  }
```

## Output

```
((true or x) and (w and x))
false
((true or (w and x)) and (w and (w and x)))
```

# And

And ::= BooleanExpression ‘&&’ BooleanExpression

```
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 BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return new And( leftOperand.replace( varName, replacement),
               rightOperand.replace( varName, replacement) );
   }

   public Object clone() {
      return new And( (BooleanExpression) leftOperand.clone( ),
               (BooleanExpression)rightOperand.clone( ) );
   }

   public String toString(){
      return "(" + leftOperand.toString() + " and " +
            rightOperand.toString() + ")";
   }
}
```

# Or

Or ::= BooleanExpression ‘or’ BooleanExpression

```
public class Or implements BooleanExpression {
   private BooleanExpression leftOperand;
   private BooleanExpression rightOperand;

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

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

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return new Or( leftOperand.replace( varName, replacement),
              rightOperand.replace( varName, replacement) );
   }

   public Object clone() {
      return new Or( (BooleanExpression) leftOperand.clone( ),
              (BooleanExpression)rightOperand.clone( ) );
   }

   public String toString() {
      return "(" + leftOperand.toString() + " or " +
            rightOperand.toString() + ")";
   }
}
```

# Not

Not ::= ‘not’ BooleanExpression

```
public class Not implements BooleanExpression {
   private BooleanExpression operand;

   public Not( BooleanExpression operand) {
      this.operand = operand;
   }

   public boolean evaluate( Context values ) {
      return  ! operand.evaluate( values );
   }

   public BooleanExpression replace( String varName,
       BooleanExpression replacement ) {
      return new Not( operand.replace( varName, replacement) );
   }

   public Object clone() {
      return new Not( (BooleanExpression) operand.clone( ) );
   }

   public String toString() {
      return "( not " +  operand.toString() + ")";
   }
```

# Constant

Constant ::= ‘true’ | ‘false’

```
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 BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      return this;
   }

   public Object clone() {
      return this;
      }

   public String toString() {
      return String.valueOf( value );
   }
}
```

# Variable

Variable ::= String

```
public class Variable implements BooleanExpression {
   private static Hashtable flyWeights = new Hashtable();

   private String name;

   public static Variable get( String name ) {
      if ( ! flyWeights.contains( name ))
         flyWeights.put( name , new Variable( name ));

      return (Variable) flyWeights.get( name );
   }

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

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

   public BooleanExpression replace( String varName,
         BooleanExpression replacement ) {
      if ( varName.equals( name ) )
         return (BooleanExpression) replacement.clone();
      else
         return this;
   }

   public Object clone() {
      return this;
   }

   public String toString() { return name; }
}
```

# Context

```
public class Context {
   Hashtable values = new Hashtable();

   public boolean getValue( String variableName ) {
      Boolean wrappedValue = (Boolean) values.get( variableName );
      return wrappedValue.booleanValue();
   }

   public void setValue( String variableName, boolean value ) {
      values.put( variableName, new Boolean( value ) );
   }
}
```

## Consequences

It's easy to change and extend the grammar

Implementing the grammar is easy

Complex grammars are hard to maintain

Adding new ways to interpret expressions

The visitor pattern is useful here

## Implementation

The pattern does not talk about parsing!

Flyweight

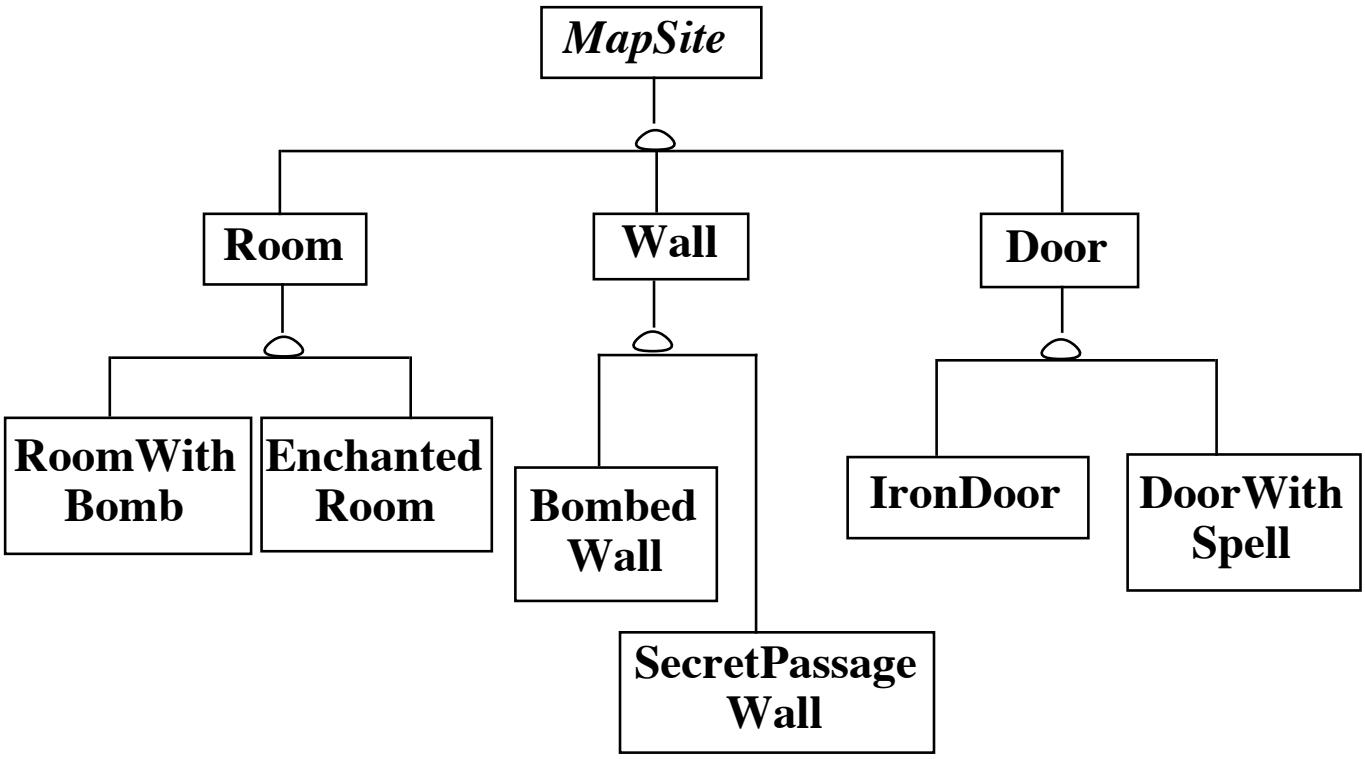
- If terminal symbols are repeated many times using the Flyweight pattern can reduce space usage
- The above example has each terminal class manage the flyweights for its objects, since Java does limited support for protecting constructors

# Factory Method

A template method for creating objects

## Example - Maze Game

## Classes for Mazes




Now a maze game has to make a maze

# Maze Class Version 1

```
class MazeGame
  {

  public Maze createMaze()
    {
    Maze aMaze = new Maze();

    Room r1 = new Room( 1 );
    Room r2 = new Room( 2 );
    Door theDoor = new Door( r1, r2);

    aMaze.addRoom( r1 );
    aMaze.addRoom( r2 );

    etc.

    return aMaze;
    }
  }
```

## How do we make other Mazes?

Subclass MazeGame, override createMaze

```
class BombedMazeGame extends MazeGame
  {

  public Maze createMaze()
    {
    Maze aMaze = new Maze();

    Room r1 = new RoomWithABomb( 1 );
    Room r2 = new RoomWithABomb( 2 );
    Door theDoor = new Door( r1, r2);

    aMaze.addRoom( r1 );
    aMaze.addRoom( r2 );

    etc.
```

Note the amount of cut and paste!

# How do we make other Mazes?

## Use Factory Method

```
class MazeGame
  {

  public Maze makeMaze() { return new Maze(); }
  public Room makeRoom(int n ) { return new Room( n ); }
  public Wall makeWall() { return new Wall(); }
  public Door makeDoor() { return new Door(); }

  public Maze CreateMaze()
    {
    Maze aMaze = makeMaze();

    Room r1 = makeRoom( 1 );
    Room r2 = makeRoom( 2 );
    Door theDoor = makeDoor( r1, r2);

    aMaze.addRoom( r1 );
    aMaze.addRoom( r2 );

    etc

    return aMaze;
    }
  }
```

Now subclass MazeGame override make methods

CreateMaze method stays the same

```
class BombedMazeGame extends MazeGame
  {

  public Room makeRoom(int n )
    {
    return new RoomWithABomb( n );
    }

  public Wall makeWall()
    {
    return new BombedWall();
    }
```

# Applicability

Use when

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- You want to localize the knowledge of which help classes is used in a class

## Consequences

- Eliminates need to hard code specific classes in code
- Requires subclassing to vary types used
- Provides hooks for subclasses
- Connects Parallel class hierarchies

# Implementation
# Two Major Varieties

- Top level Factory method is in an abstract class

```
abstract class MazeGame
   {
   public Maze makeMaze();
   public Room makeRoom(int n );
   public Wall makeWall();
   public Door makeDoor();
   etc.
   }

class MazeGame
   {
   public:
      virtual Maze* makeMaze() = 0;
      virtual Room* makeRoom(int n ) = 0;
      virtual Wall* makeWall() = 0;
      virtual Door* makeDoor() = 0;
```

- Top level Factory method is in a concrete class

  See examples on previous slides

# Implementation - Continued
# Parameterized Factory Methods

Let the factory method return multiple products

```
class Hershey
  {

  public Candy makeChocolateStuff( CandyType id )
    {
    if ( id == MarsBars ) return new MarsBars();
    if ( id == M&Ms ) return new M&Ms();
    if ( id == SpecialRich ) return new SpecialRich();

    return new PureChocolate();
    }

class GenericBrand extends Hershey
  {
  public Candy makeChocolateStuff( CandyType id )
    {
    if ( id == M&Ms ) return new Flupps();
    if ( id == Milk ) return new MilkChocolate();
    return super.makeChocolateStuff();
    }
  }
```

# C++ Templates to Avoid Subclassing

```
template <class ChocolateType>
class Hershey
   {
   public:
      virtual Candy* makeChocolateStuff( );
   }

template <class ChocolateType>
Candy* Hershey<ChocolateType>::makeChocolateStuff( )
   {
   return new ChocolateType;
   }

Hershey<SpecialRich> theBest;
```

## Java forName and Factory methods

With Java's reflection you can use a Class or a String to specify which type of object to create

Using a string replaces compile checks with runtime errors

```
class Hershey
  {
  private String chocolateType;

  public Hershey( String chocolate )
    {
    chocolateType = chocolate;
    }

  public Candy makeChocolateStuff( )
    {
    Class candyClass =  Class.forName( chocolateType );
    return (Candy) candyClass.newInstance();
    }


Hershey theBest = new Heshsey( "SpecialRich" );
```

# Clients Can Use Factory Methods

```
class CandyStore
  {
  Hershey supplier;
  public restock()
    {
    blah

    if ( chocolateStock.amount() < 10 )
      {
      chocolateStock.add(
        supplier.makeChocolateStuff() );
      }

  blah
```