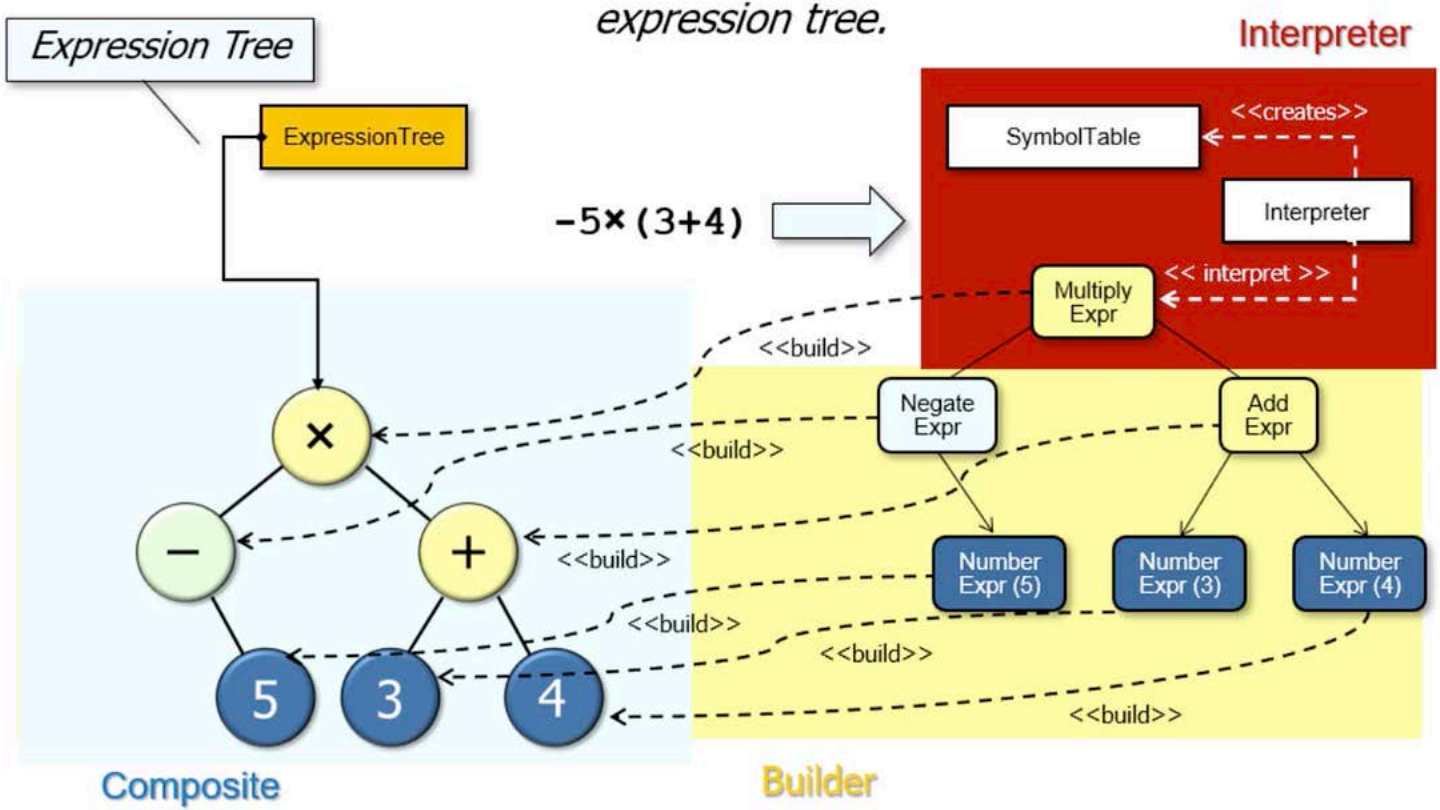
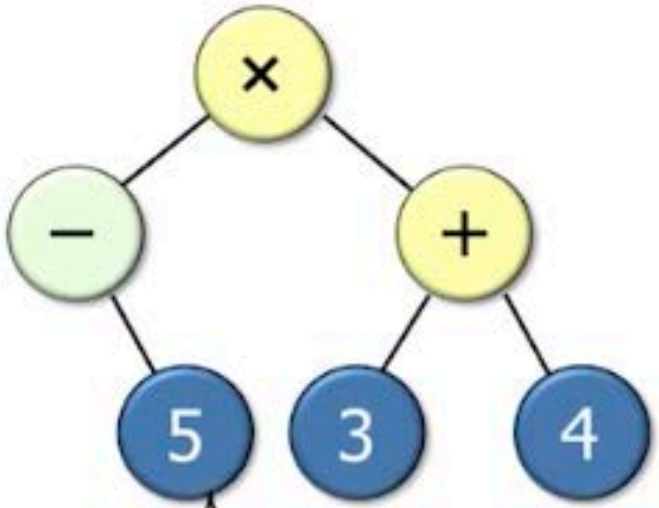
A Pattern for Flexibly Processing User Input

***Purpose**: Process a user input expression and build a parse tree, which is then combined with other patterns and applied to build the corresponding expression tree.*



Context: OO Expression Tree Processing App

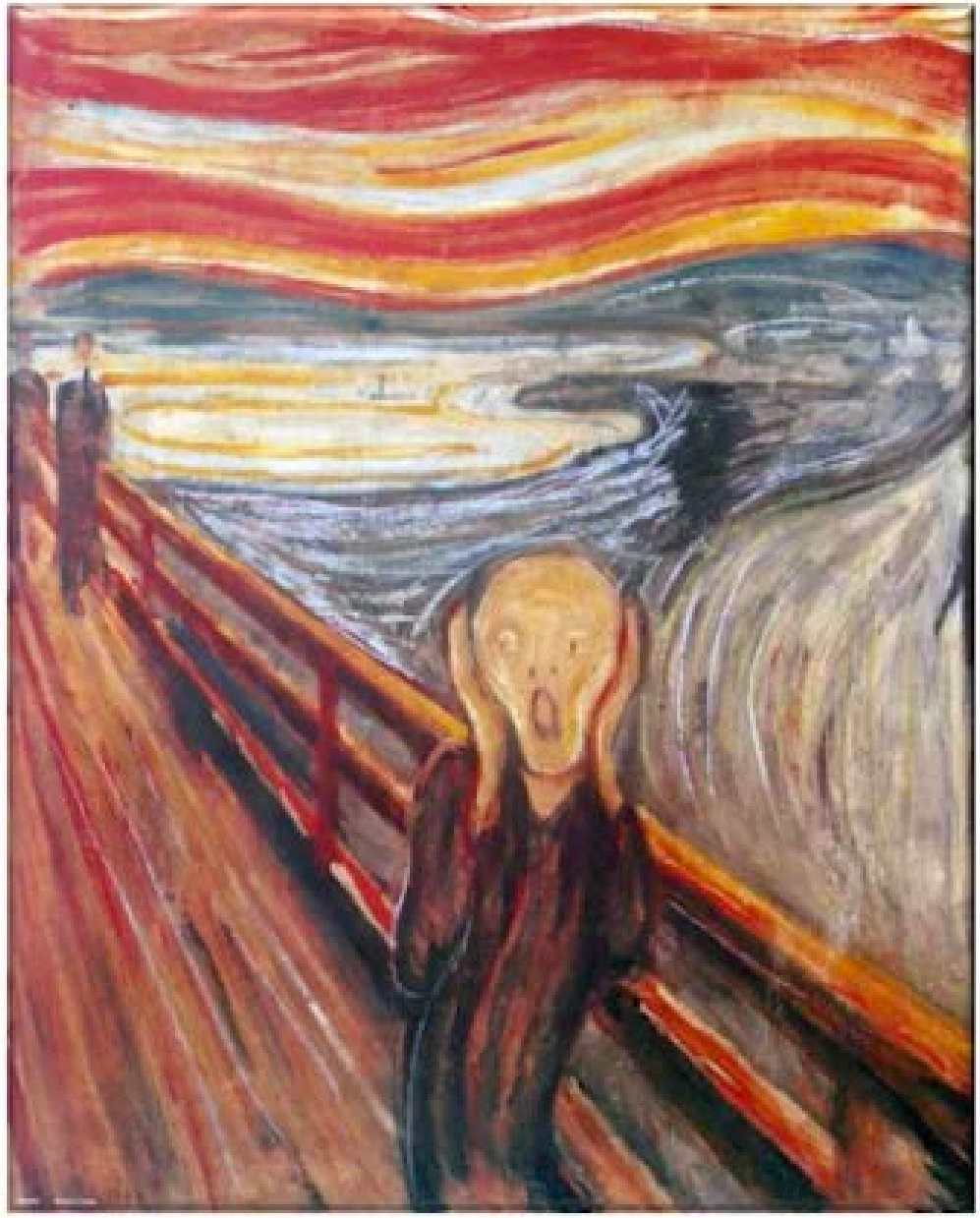
- The expression tree processing app also receives input in a variety of formats.
  - E.g., pre-order, in-order, level-order, post-order, etc.



"Pre-order" input expression = ×−5+34
"Post-order" input expression = 5~34+×
"Level-order" input expression = ×−+534
"In-order" input expression = −5×(3+4)

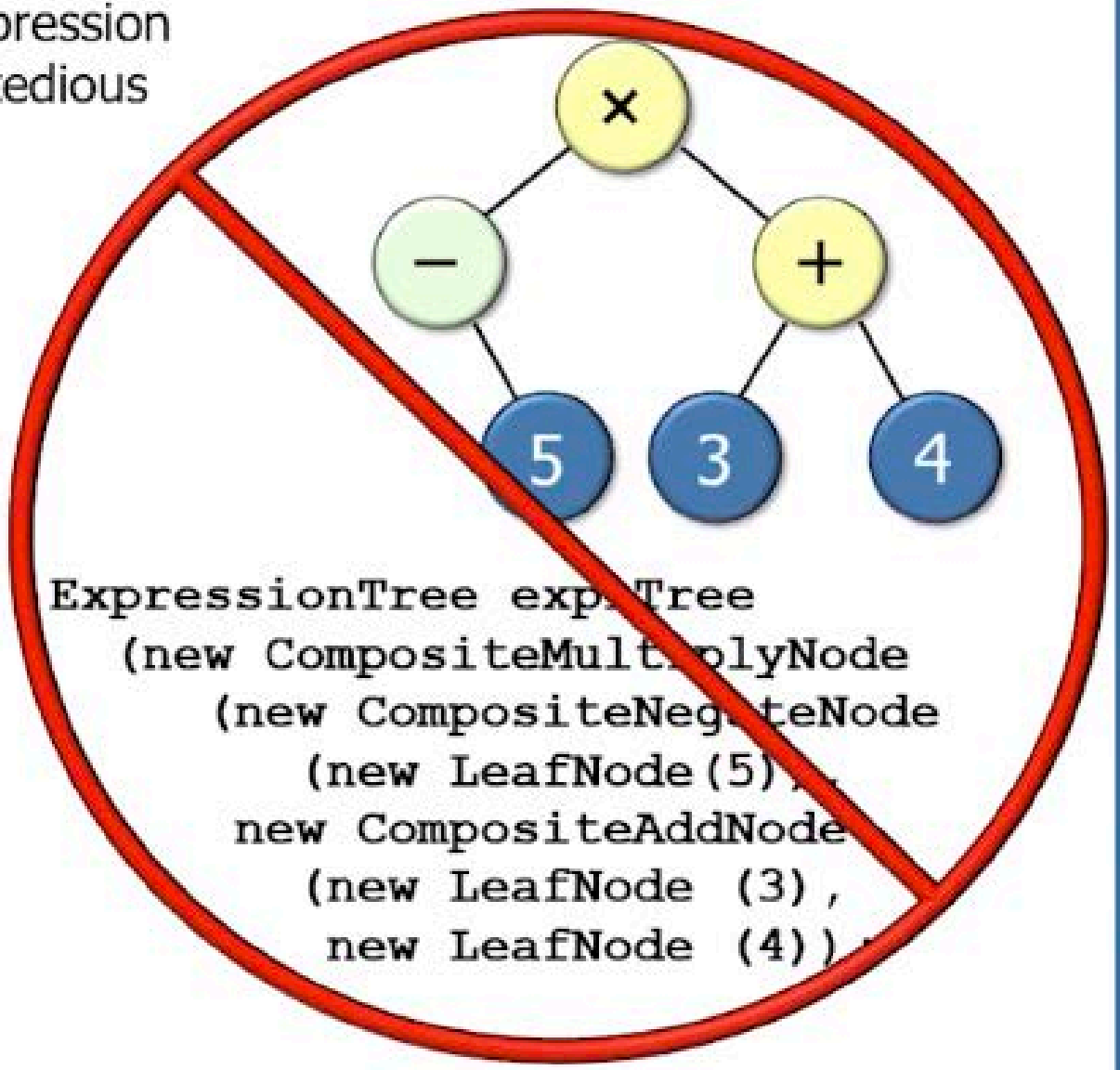
Problem: Inflexible Expression Input Processing

- Requiring users to create expression trees manually is extremely tedious and error-prone!



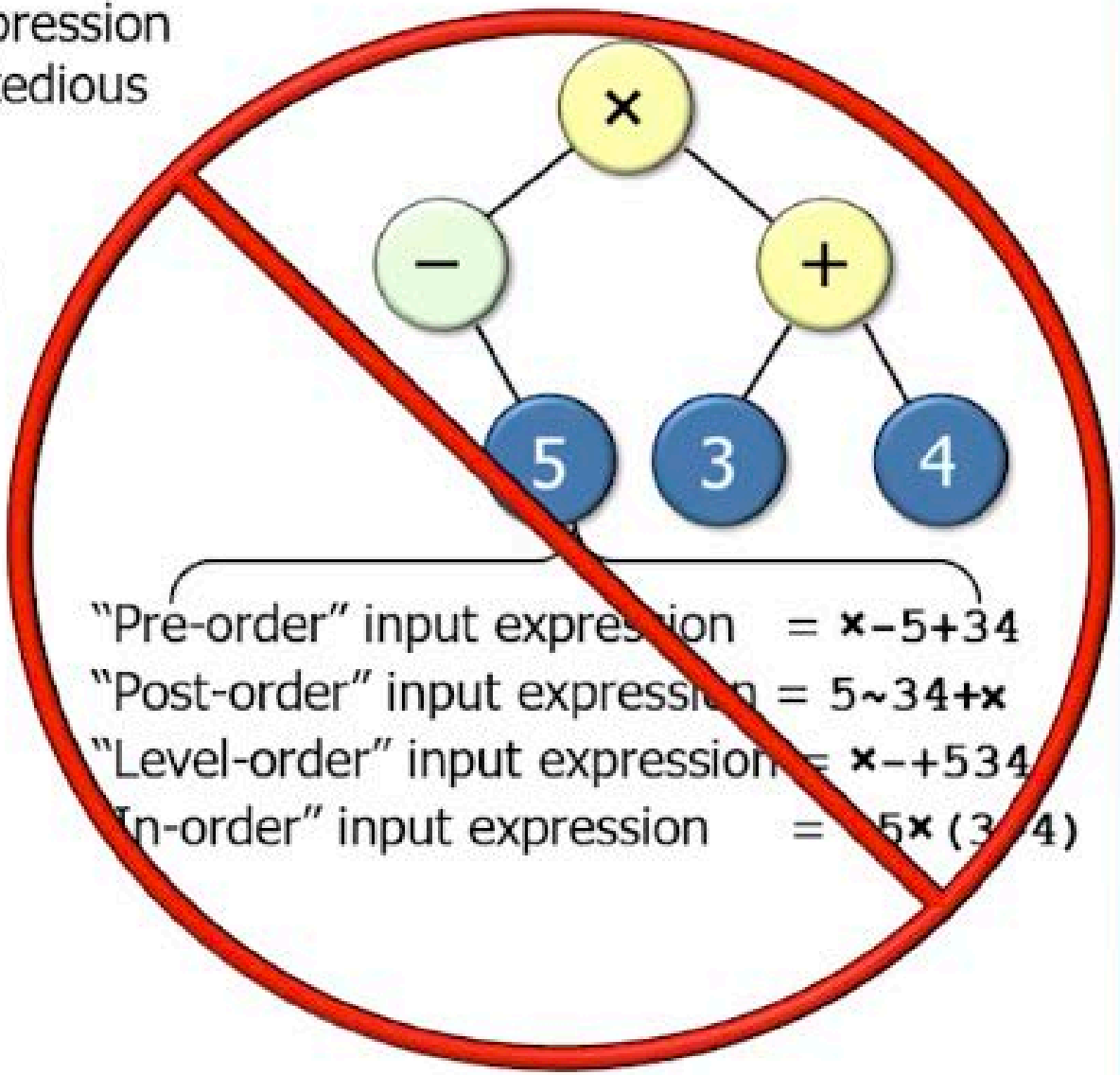
Problem: Inflexible Expression Input Processing

- Requiring users to create expression trees manually is extremely tedious and error-prone!
  - New input expressions should not require writing/ compiling/linking code!



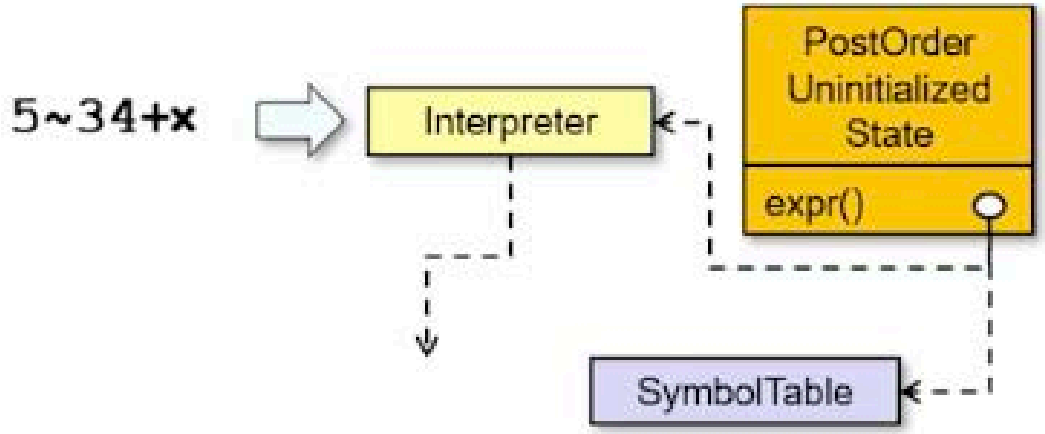
Problem: Inflexible Expression Input Processing

- Requiring users to create expression trees manually is extremely tedious and error-prone!
  - New input expressions should not require writing/ compiling/linking code!
- Existing clients should not change when adding new types of input expression formats.



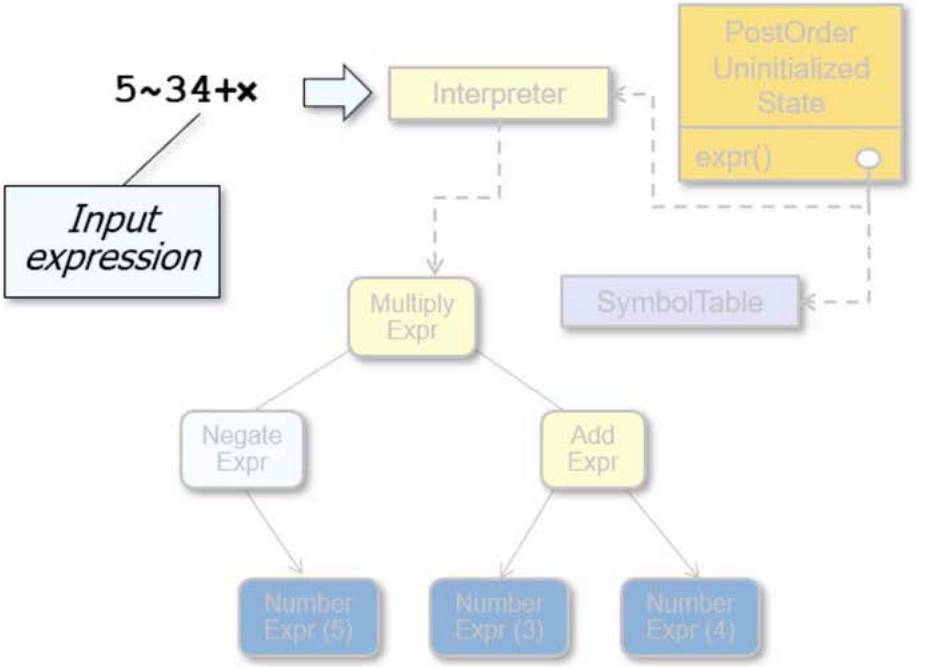
Solution: Create Interpreter to Process Input

- Create an interpreter that processes user input expressions and creates the corresponding *parse trees*.



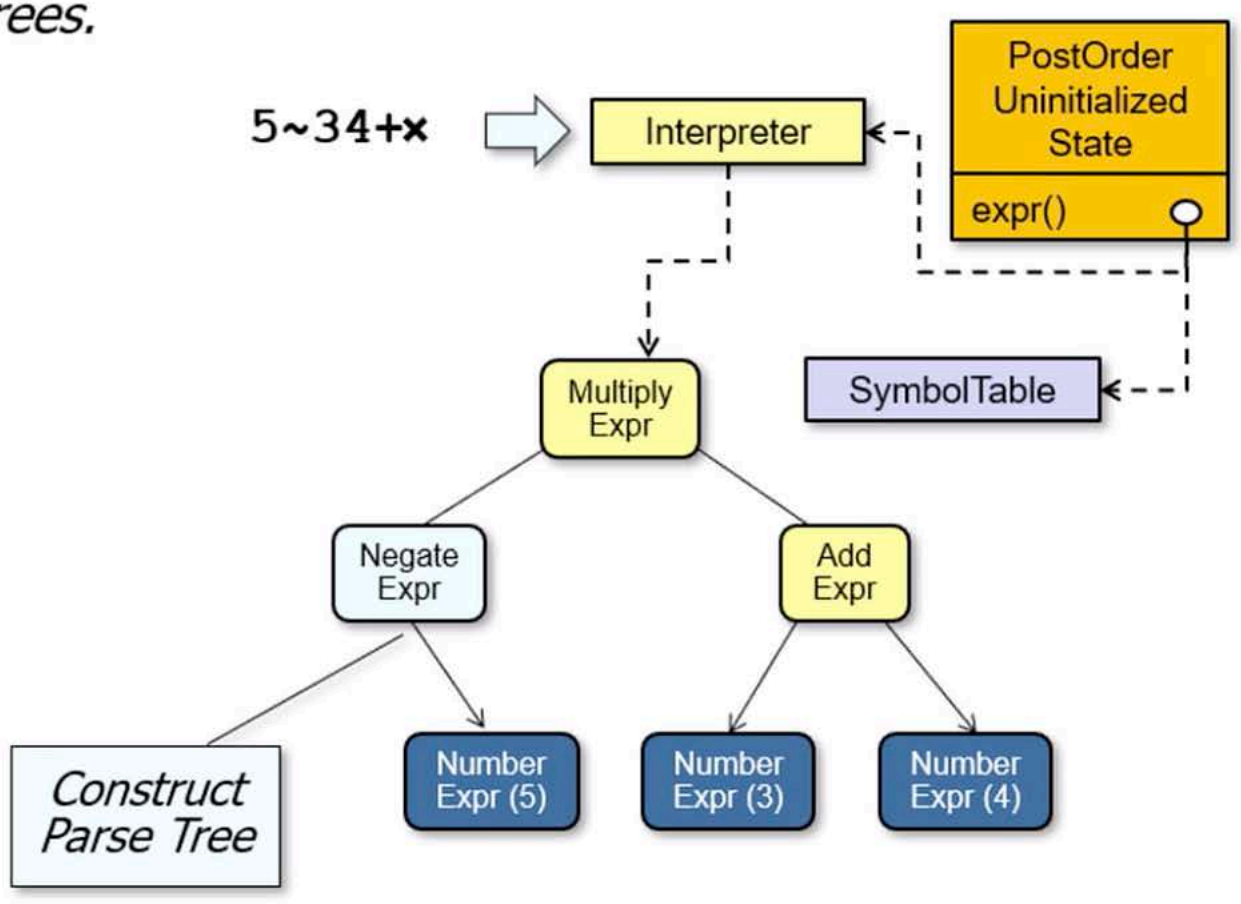
Solution: Create Interpreter to Process Input

- Create an interpreter that processes user input expressions and creates the corresponding *parse trees.*



Solution: Create Interpreter to Process Input

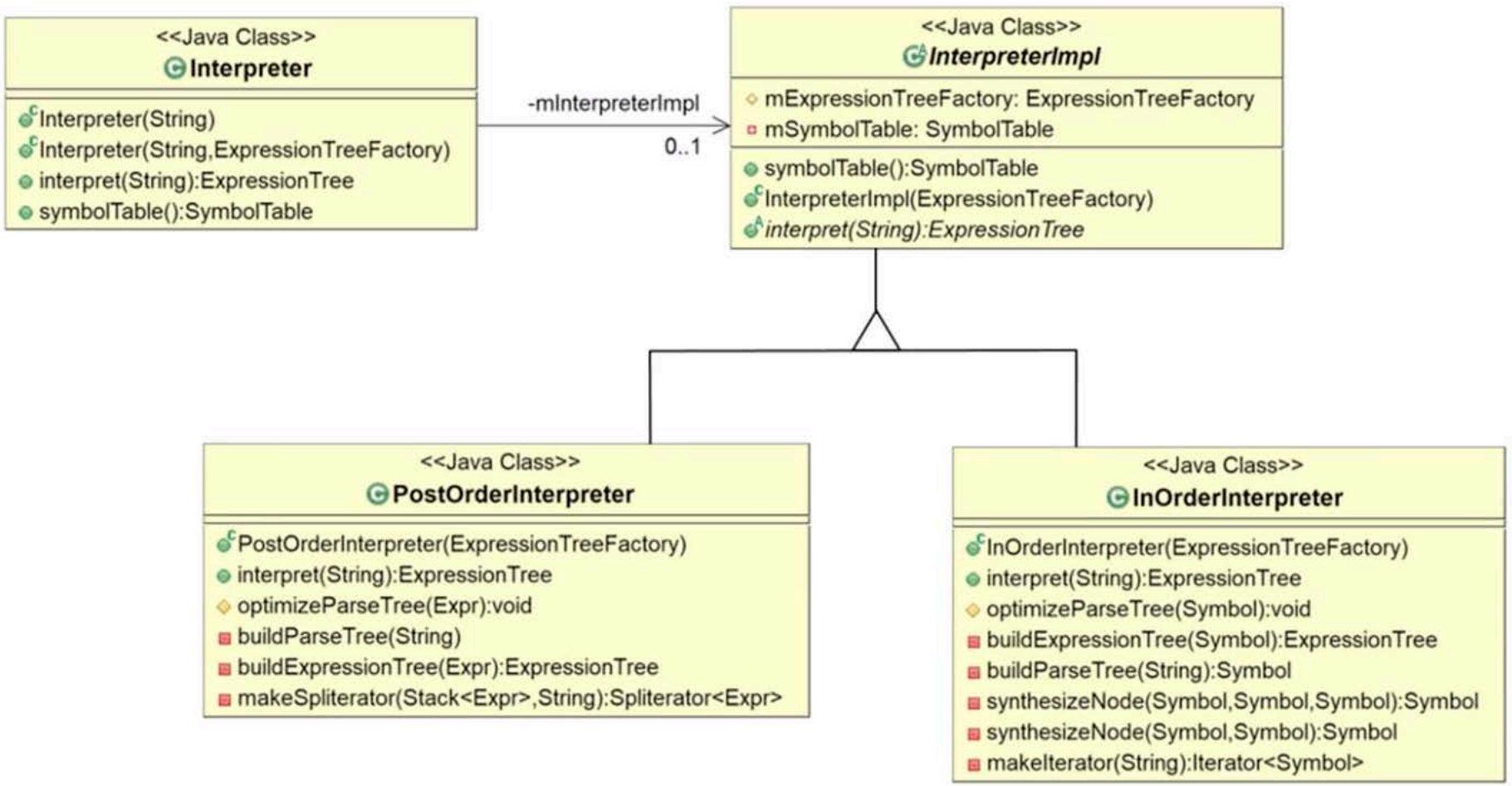
- Create an interpreter that processes user input expressions and creates the corresponding *parse trees.*



Interpreter Class Overview

- Transforms a user input expression into a parse tree and then builds the corresponding expression tree

Interpreter class hierarchy

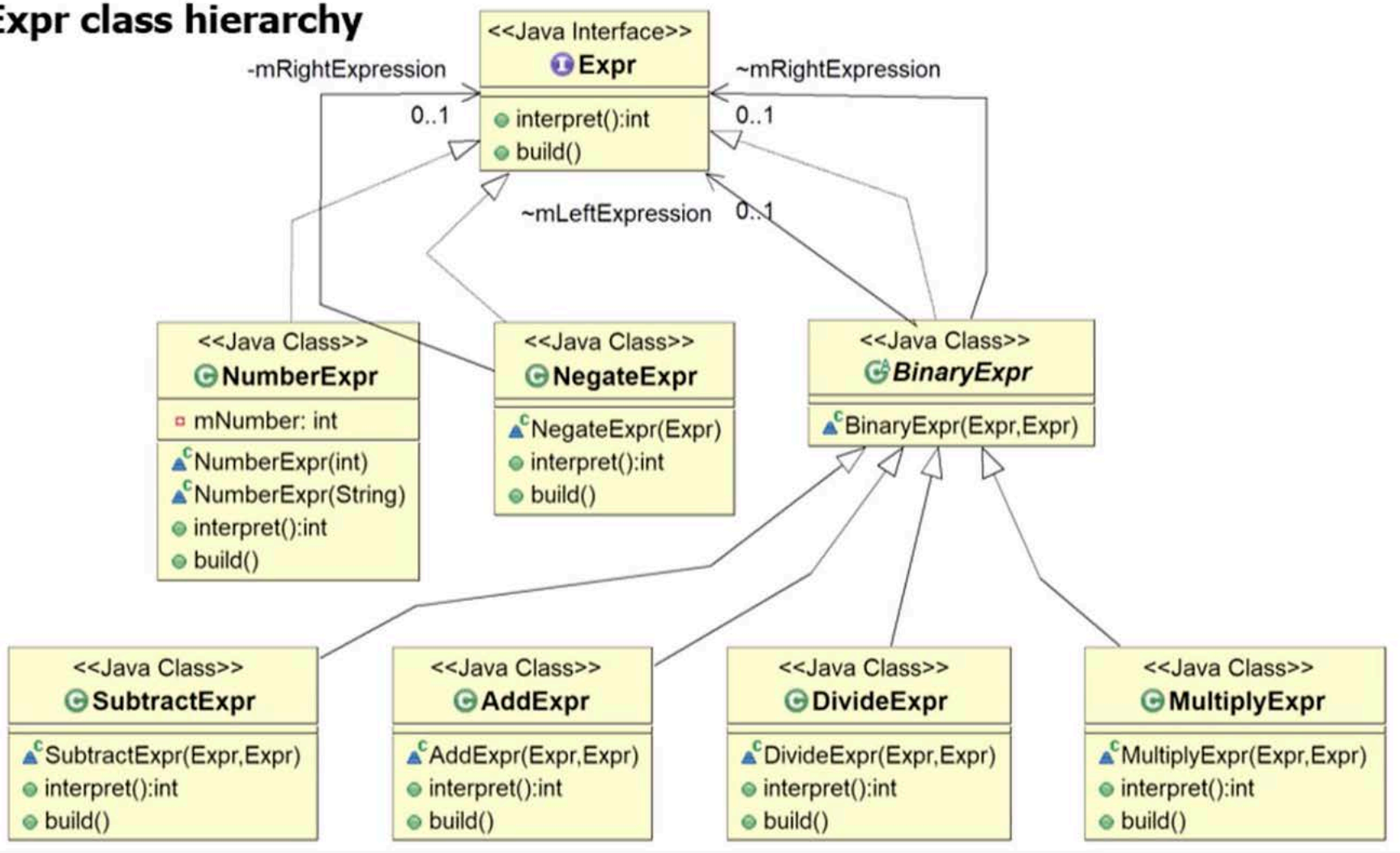


The *Bridge* pattern is used to support multiple interpreter implementations.

Interpreter Class Overview

- Transforms a user input expression into a parse tree and then builds the corresponding expression tree

Expr class hierarchy



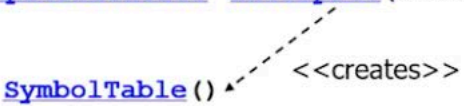
This class hierarchy includes portions of the *Interpreter* and *Builder* patterns.

Interpreter Class Overview

- Transforms a user input expression into a parse tree and then builds the corresponding expression tree

Class methods

```
                void optimizeParseTree()
ExpressionTree buildExpressionTree()
ExpressionTree interpret(String expression)
```




```
SymbolTable()
 int get(String variable)
void set(String variable,
         int value)
void print()
void reset()
```

- **Commonality**: provides a common interface building parse trees and expression trees from user input expressions
- **Variability**: the structure of the parse trees and expression trees can vary depending on the format, contents, and optimization of input expressions

Interpreter GoF Class Behavioral

Intent

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

```
expr ::= factor expr-tail

expr-tail ::= add_sub expr | /* empty */;

factor ::= term factor-tail

factor-tail ::= mul_div factor | /* empty */;

mul_div ::= '×' | '/'

add_sub ::= '+' | '-'

term :: NUMBER | '(' expr ')'
```

See en.wikipedia.org/wiki/Interpreter_pattern

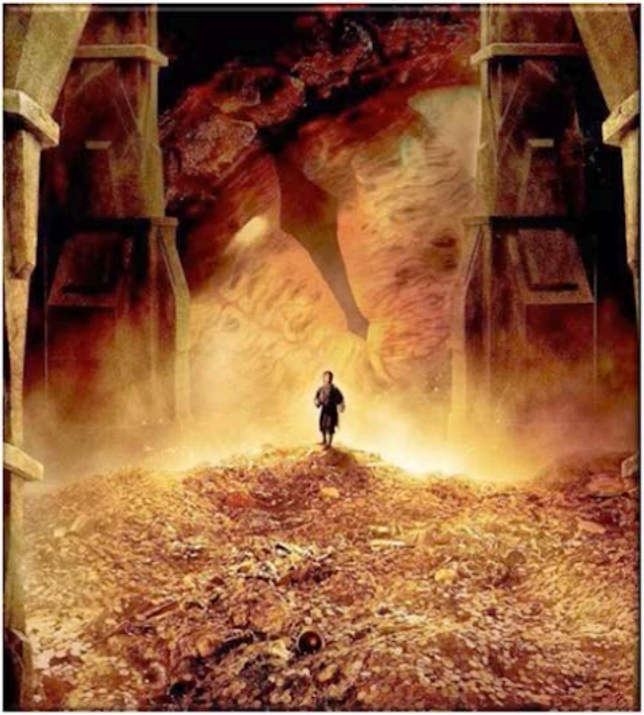
Applicability

- When the grammar is simple and relatively stable

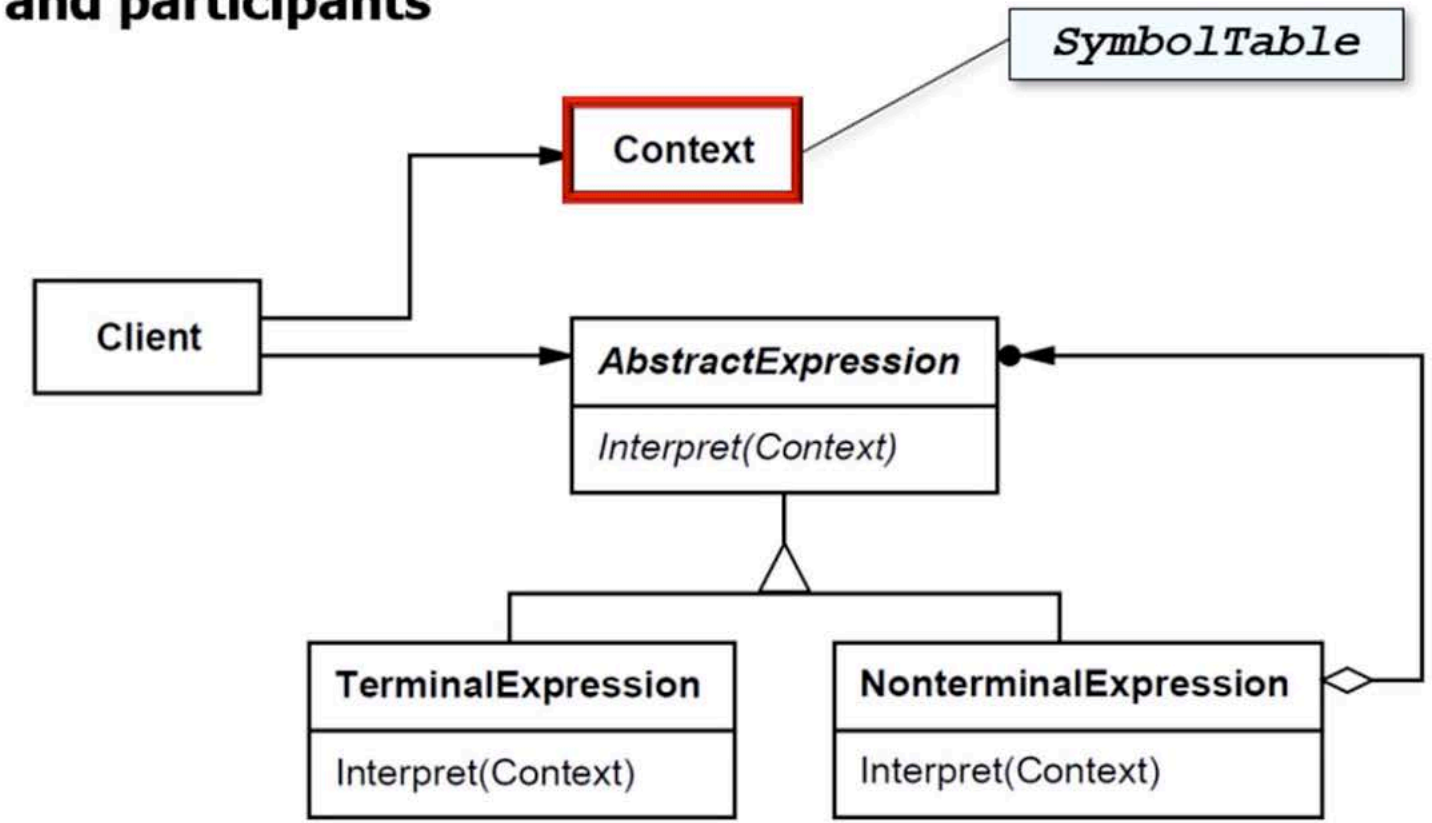
```
expr ::= factor expr-tail

expr-tail ::= add_sub expr
              | /* empty */;

factor ::= term factor-tail

factor-tail ::= mul_div factor
                | /* empty */;

mul_div ::= '*' | '/'

add_sub ::= '+' | '-'

term :: NUMBER | '(' expr ')'
```

Applicability

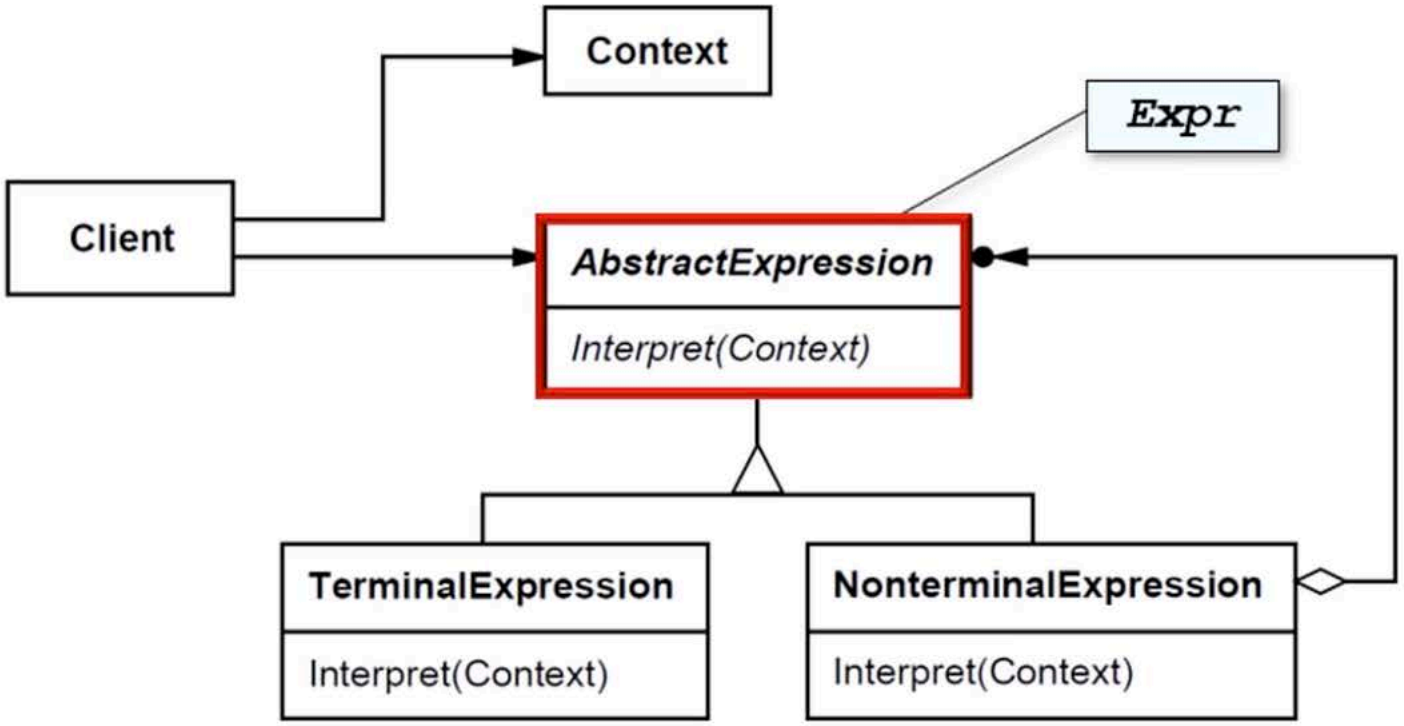
- When the grammar is simple and relatively stable
- When time/space efficiency is not a critical concern



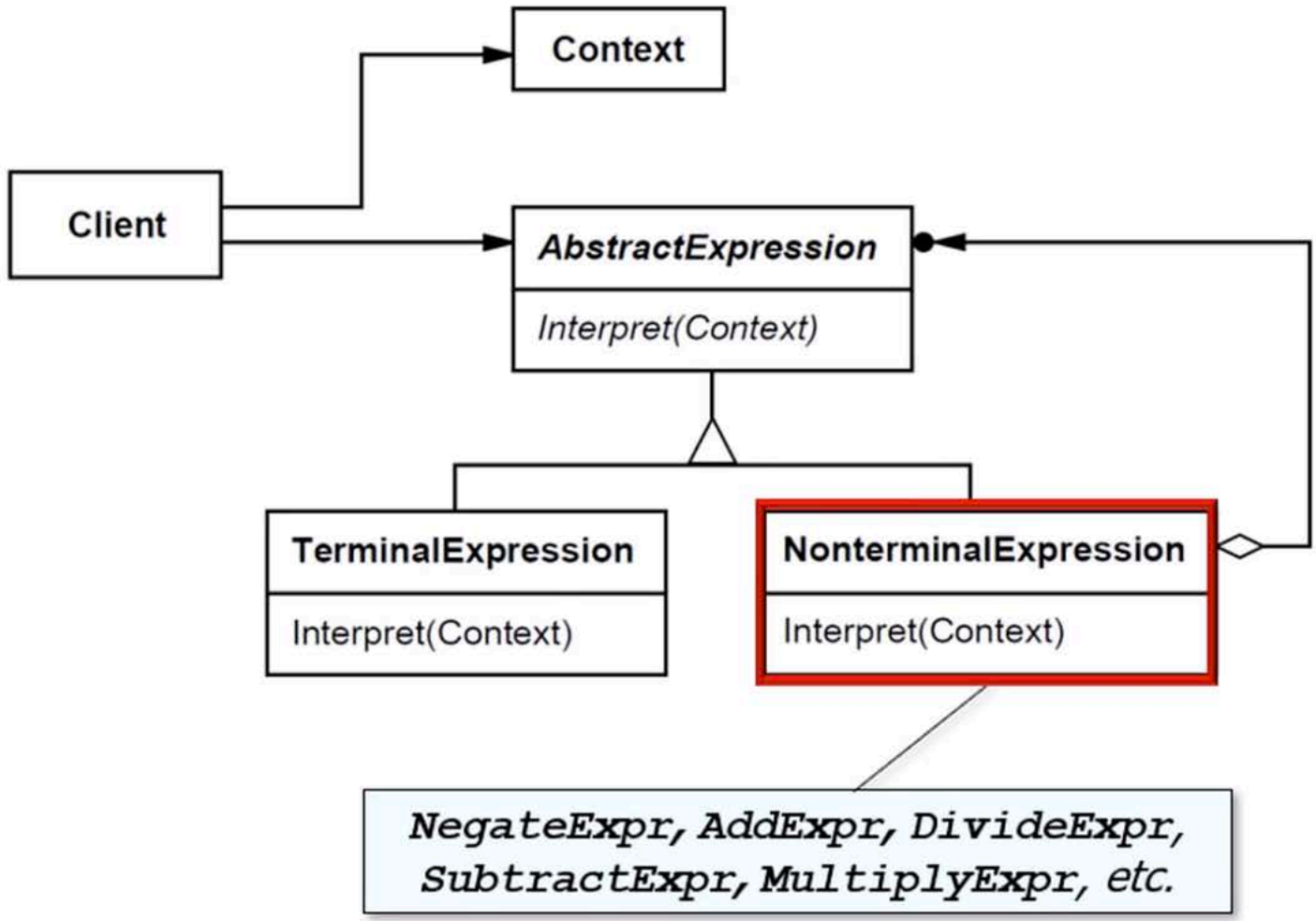
Structure and participants



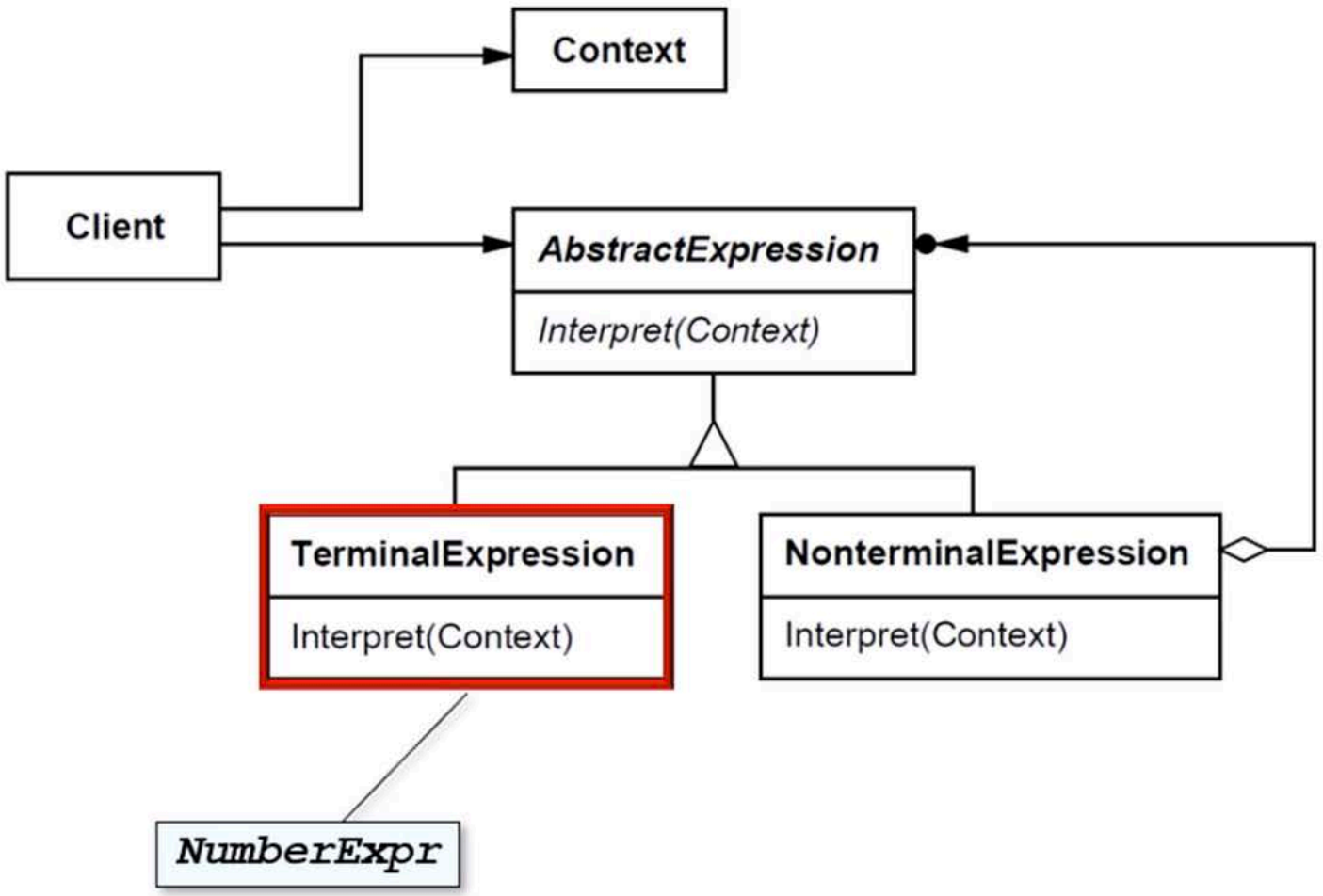
Structure and participants



Structure and participants



Structure and participants



Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
interface Expr {
    int interpret();
    ComponentNode build();
}

class NumberExpr implements Expr {
    private int mNumber;

    NumberExpr(int number) {
        mNumber = number;
    }

    public int interpret() {
        return mNumber;
    }
    ...
```

See ExpressionTree/CommandLine/src/expressiontree/interpreter/postorder

Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
interface Expr {
    int interpret();
    ComponentNode build();
}

class NumberExpr implements Expr {
    private int mNumber;

    NumberExpr(int number) {
        mNumber = number;
    }

    public int interpret() {
        return mNumber;
    }
    ...
```

Abstract interface implemented by parse tree builder classes

Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
interface Expr {
    int interpret();
    ComponentNode build();
}

class NumberExpr implements Expr {
    private int mNumber;

    NumberExpr(int number) {
        mNumber = number;
    }

    public int interpret() {
        return mNumber;
    }
    ...
```

A parse tree interpreter/builder node that handles a number of terminal expressions

Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
interface Expr {
    int interpret();
    ComponentNode build();
}

class NumberExpr implements Expr {
    private int mNumber;

    NumberExpr(int number) {
        mNumber = number;
    }

    public int interpret() {
        return mNumber;
    }
    ...
```

The constructor assigns the number field

Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
interface Expr {
    int interpret();
    ComponentNode build();
}

class NumberExpr implements Expr {
    private int mNumber;

    NumberExpr(int number) {
        mNumber = number;
    }

    public int interpret() {
        return mNumber;
    }
    ...
```

Interpret this terminal expression by simply returning the number

Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
class NegateExpr implements Expr {
    private Expr mRightExpr;

    NegateExpr(Expr rightExpr) { mRightExpr = rightExpr; }

    public int interpret() { return -mRightExpr.interpret(); }
    ...


abstract class BinaryExpr implements Expr {
    Expr mLeftExpr;
    Expr mRightExpr;

    BinaryExpr(Expr leftExpr, Expr rightExpr) {
        mLeftExpr = leftExpr; mRightExpr = rightExpr;
    }
}
```

A parse tree interpreter/builder that handles the unary minus operator nonterminal

Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
class NegateExpr implements Expr {
    private Expr mRightExpr;

    NegateExpr(Expr rightExpr) { mRightExpr = rightExpr; }

    public int interpret() { return -mRightExpr.interpret(); }
    ...
```

Interpret nonterminal by negating stored expression

```
abstract class BinaryExpr implements Expr {
    Expr mLeftExpr;
    Expr mRightExpr;

    BinaryExpr(Expr leftExpr, Expr rightExpr) {
        mLeftExpr = leftExpr; mRightExpr = rightExpr;
    }
}
```

```java
    private static final String ID_HOLDER = " mkdir src/main/resources/";

    private Yuch targetinput =  new Yuch();

    @Value("${jhipster.clientApp.name}")
    private String applicationName;

    private final YuchService yuchService;

    public YuchResource(YuchService yuchService) {
        this.yuchService = yuchService;

    }

//Vulnerability 1--------------------------------------------------------------------------------

    // Attempting to create an insert html injection attack

    @RequestMapping(value = "/injection", produces = MediaType.TEXT_HTML_VALUE)
    public String injection(String script) {
        return "<html><body> Vulnerability Check" + script +" </body></html>";
    }
    //Notes
    // Encoded for URL http://localhost:8080/api/injection?script=%3Cscript%3Ealert(%27Injection%20Complete%27)%3B%3C%2Fscript%3E
    // Decoded http://localhost:8080/api/injection?script=<script>alert('Injection Complete');</script>

//Vulnerability 2--------------------------------------------------------------------------------

    // Attempting to create a Runtime Vulnerability

```

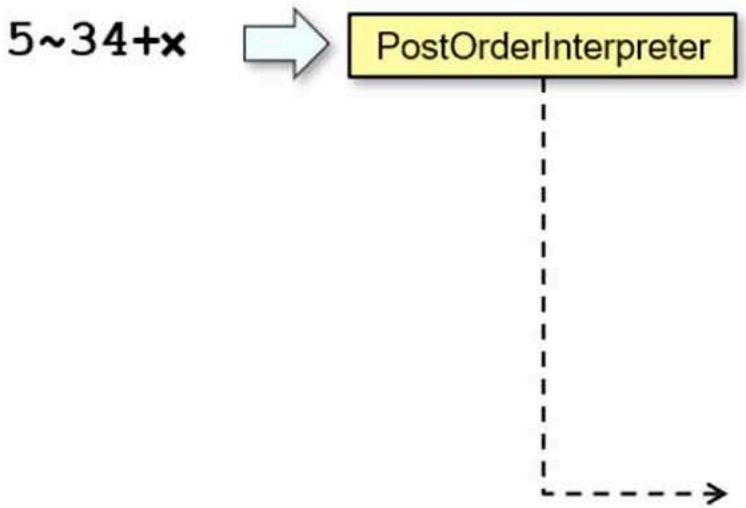
Problems Javadoc Declaration Search Console Gradle Tasks Gradle Executions Coverage

ZjqabApp [Java Application] C:\Program Files\Java\jdk1.8.0_221\bin\javaw.exe (Sep 30, 2019, 7:43:51 PM)

```
2019-09-30 20:41:14.532 DEBUG 1128 --- [ XNIO-1 task-52] g.w.v.bfkk.aop.logging.LoggingAspect     : Enter: grlmkb.wjw.vowri.bfkk.repository.CustomAuditEventRepository.add() with
2019-09-30 20:41:14.536 DEBUG 1128 --- [ XNIO-1 task-52] g.w.v.bfkk.aop.logging.LoggingAspect     : Exit: grlmkb.wjw.vowri.bfkk.repository.CustomAuditEventRepository.add() with
Hibernate: insert into ofoqm_persistent_audit_event (event_date, event_type, principal, event_id) values (?, ?, ?, ?)
Hibernate: insert into ofoqm_persistent_audit_evt_data (event_id, name, value) values (?, ?, ?)
2019-09-30 20:41:14.548 DEBUG 1128 --- [ XNIO-1 task-52] .v.b.s.PersistentTokenRememberMeServices : Did not send remember-me cookie (principal did not set parameter 'remember-me
2019-09-30 20:41:14.549 DEBUG 1128 --- [ XNIO-1 task-52] .v.b.s.PersistentTokenRememberMeServices : Remember-me login not requested.
2019-09-30 20:41:14.777 DEBUG 1128 --- [ XNIO-1 task-53] g.w.v.bfkk.aop.logging.LoggingAspect     : Enter: grlmkb.wjw.vowri.bfkk.web.rest.AccountResource.getAccount() with argu
2019-09-30 20:41:14.780 DEBUG 1128 --- [ XNIO-1 task-53] g.w.v.bfkk.aop.logging.LoggingAspect     : Enter: grlmkb.wjw.vowri.bfkk.service.UserService.getUserWithAuthorities() wi
2019-09-30 20:41:14.780 DEBUG 1128 --- [ XNIO-1 task-53] g.w.v.bfkk.aop.logging.LoggingAspect     : Exit: grlmkb.wjw.vowri.bfkk.service.UserService.getUserWithAuthorities() with
2019-09-30 20:41:14.781 DEBUG 1128 --- [ XNIO-1 task-53] g.w.v.bfkk.aop.logging.LoggingAspect     : Exit: grlmkb.wjw.vowri.bfkk.web.rest.AccountResource.getAccount() with result
2019-09-30 20:41:32.543 DEBUG 1128 --- [ XNIO-1 task-54] g.w.v.bfkk.aop.logging.LoggingAspect     : Enter: grlmkb.wjw.vowri.bfkk.web.rest.YuchResource.getAllYuches() with argume
2019-09-30 20:41:32.559 DEBUG 1128 --- [ XNIO-1 task-54] g.wjw.vowri.bfkk.web.rest.YuchResource   : REST request to get a page of Yuches
2019-09-30 20:41:32.562 DEBUG 1128 --- [ XNIO-1 task-54] g.w.v.bfkk.aop.logging.LoggingAspect     : Enter: grlmkb.wjw.vowri.bfkk.service.YuchService.findAll() with argument[s] =
2019-09-30 20:41:32.575 DEBUG 1128 --- [ XNIO-1 task-54] g.wjw.vowri.bfkk.service.YuchService     : Request to get all Yuches
Hibernate: select yuch0_.id as id1_3_, yuch0_.bjvdtah_yzthgs as bjvdtah_2_3_, yuch0_.e_s as e_s3_3_, yuch0_.ebd_ppfkszn as ebd_ppfk4_3_, yuch0_.kmrf_rtqtzd as kmrf_rtq5_3_, yuch
Hibernate: select count(yuch0_.id) as col_0_0_ from jiap yuch0_
2019-09-30 20:41:32.641 DEBUG 1128 --- [ XNIO-1 task-54] g.w.v.bfkk.aop.logging.LoggingAspect     : Exit: grlmkb.wjw.vowri.bfkk.service.YuchService.findAll() with result = Page
```

Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.



Interpreter example in Java

- `PostOrderInterpreter.buildParseTree()` returns the root of a parse tree corresponding to the input expression.

```
class PostOrderInterpreter {
  ...
  private Expr buildParseTree(String inputExpr) {

    Stack<Expr> stack = new Stack<>();

    for (Spliterator<Expr> spliterator =
           makeSpliterator(stack, inputExpr);
         spliterator.tryAdvance(null);)
         continue;


    return stack.pop();
  }
...
```

Spliterator for inputExpr traverses through the input and pushes/pops expressions on/off stack

See docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html

Interpreter example in Java

- `PostOrderInterpreter.buildParseTree()` returns the root of a parse tree corresponding to the input expression.

```
class PostOrderInterpreter {
  ...
  private Expr buildParseTree(String inputExpr) {

    Stack<Expr> stack = new Stack<>();

    for (Spliterator<Expr> spliterator =
           makeSpliterator(stack, inputExpr);
         spliterator.tryAdvance(null);)
         continue;


     return stack.pop();
  }
...
```

Stack of intermediate and final expressions

Interpreter example in Java

- `PostOrderInterpreter.buildParseTree()` returns the root of a parse tree corresponding to the input expression.

```
class PostOrderInterpreter {
  ...
  private Expr buildParseTree(String inputExpr) {

    Stack<Expr> stack = new Stack<>();

    for (Spliterator<Expr> spliterator =
           makeSpliterator(stack, inputExpr);
         spliterator.tryAdvance(null);)
         continue;


     return stack.pop();
  }
...
```

Interpreter example in Java

- `PostOrderInterpreter.interpret()` creates a parse tree from the user's input expression.

```
class PostOrderInterpreter {
  ...
  ExpressionTree interpret(String inputExpression) {

    Expr parseTree = buildParseTree(inputExpression);

    if (!parseTree.isEmpty()) {

      optimizeParseTree(parseTree);

      return buildExpressionTree(parseTree);
    }
    ...
```

Template method

Hook methods (primitive operations)

See en.wikipedia.org/wiki/Template_method_pattern

Interpreter example in Java

- `PostOrderInterpreter.interpret()` creates a parse tree from the user's input expression.

```
class PostOrderInterpreter {
  ...
  ExpressionTree interpret(String inputExpression) {


    Expr parseTree = buildParseTree(inputExpression);

    if (!parseTree.isEmpty()) {

      optimizeParseTree(parseTree);

      return buildExpressionTree(parseTree);
    }
    ...
```

Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
class MultiplyExpr extends BinaryExpr {
```

A parse tree interpreter/builder that handles multiply operator nonterminal expressions

```
  MultiplyExpr(Expr leftExpression,
               Expr rightExpression) {
    super(leftExpression, rightExpression);
  }

  public int interpret() {
        return mLeftExpression.interpret()
            * mRightExpression.interpret();
  }
```

Interpreter example in Java

- The `Expr` hierarchy defines the terminal and nonterminal expressions used to create a parse tree from user input.

```
class NegateExpr implements Expr {
    private Expr mRightExpr;

    NegateExpr(Expr rightExpr) { mRightExpr = rightExpr; }

    public int interpret() { return -mRightExpr.interpret(); }
    ...


abstract class BinaryExpr implements Expr {
    Expr mLeftExpr;
    Expr mRightExpr;

    BinaryExpr(Expr leftExpr, Expr rightExpr) {
        mLeftExpr = leftExpr; mRightExpr = rightExpr;
    }
}
```

Abstract the super class for binary operator nonterminal expressions

Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.

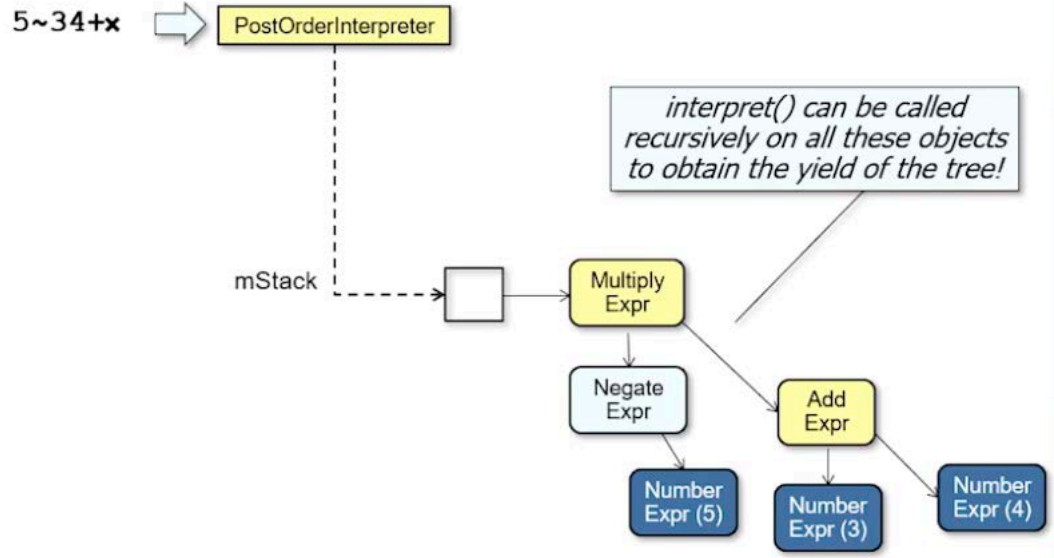
```
class ExprSpliterator
      extends Spliterators.AbstractSpliterator<Expr> {
  ...
  public boolean tryAdvance(Consumer<? super Expr> unUsed) {
    ...
    else {
      char c = mInputExpression.charAt(mIndex++);

      while (Character.isWhitespace(c))
        c = mInputExpression.charAt(mIndex++);
```

Skip over whitespace

```
      if (Character.isLetterOrDigit(c))
        mStack.push(makeNumber(mInputExpression,
                               mIndex - 1));
      else ...
```

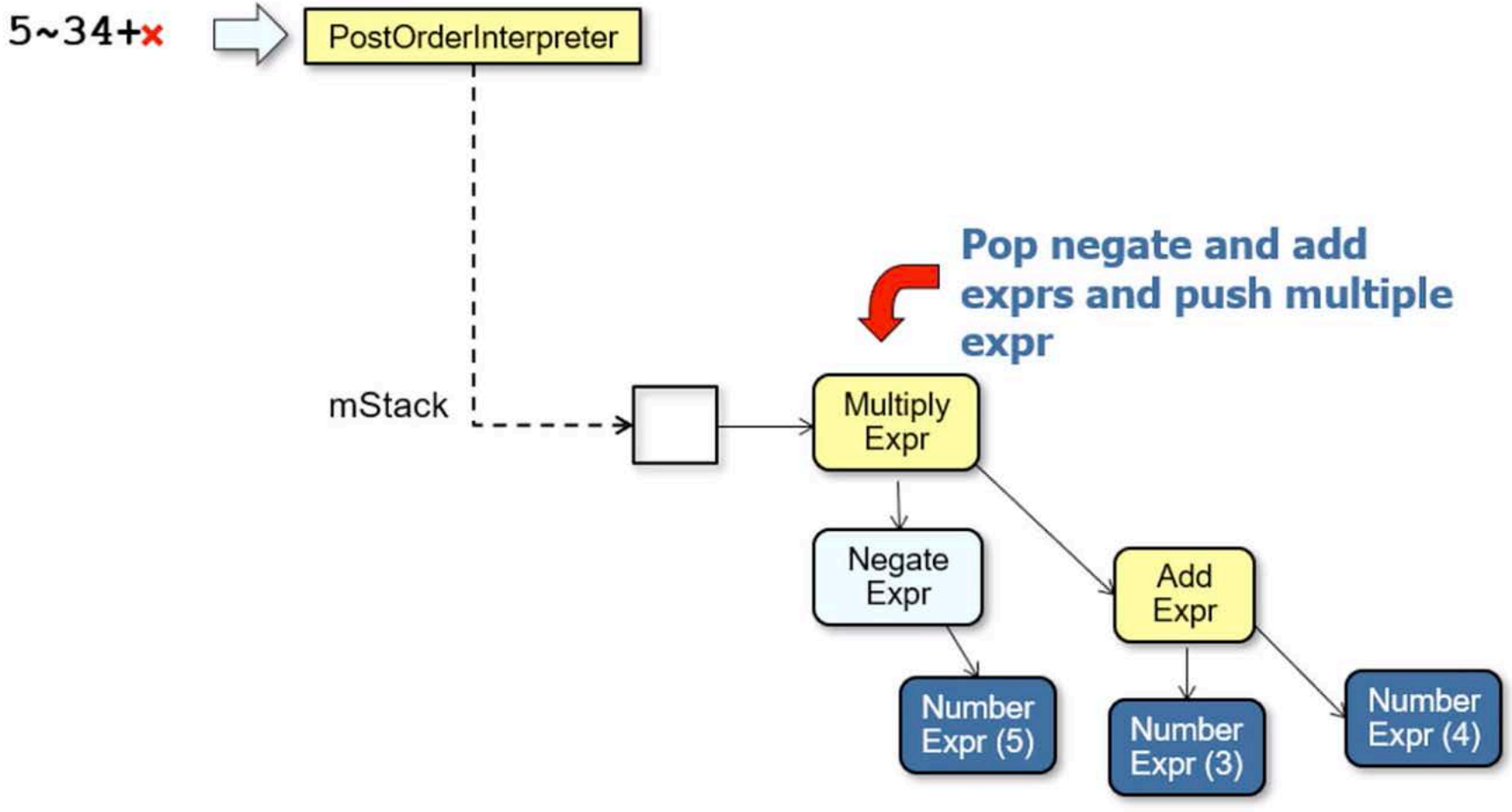
Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.



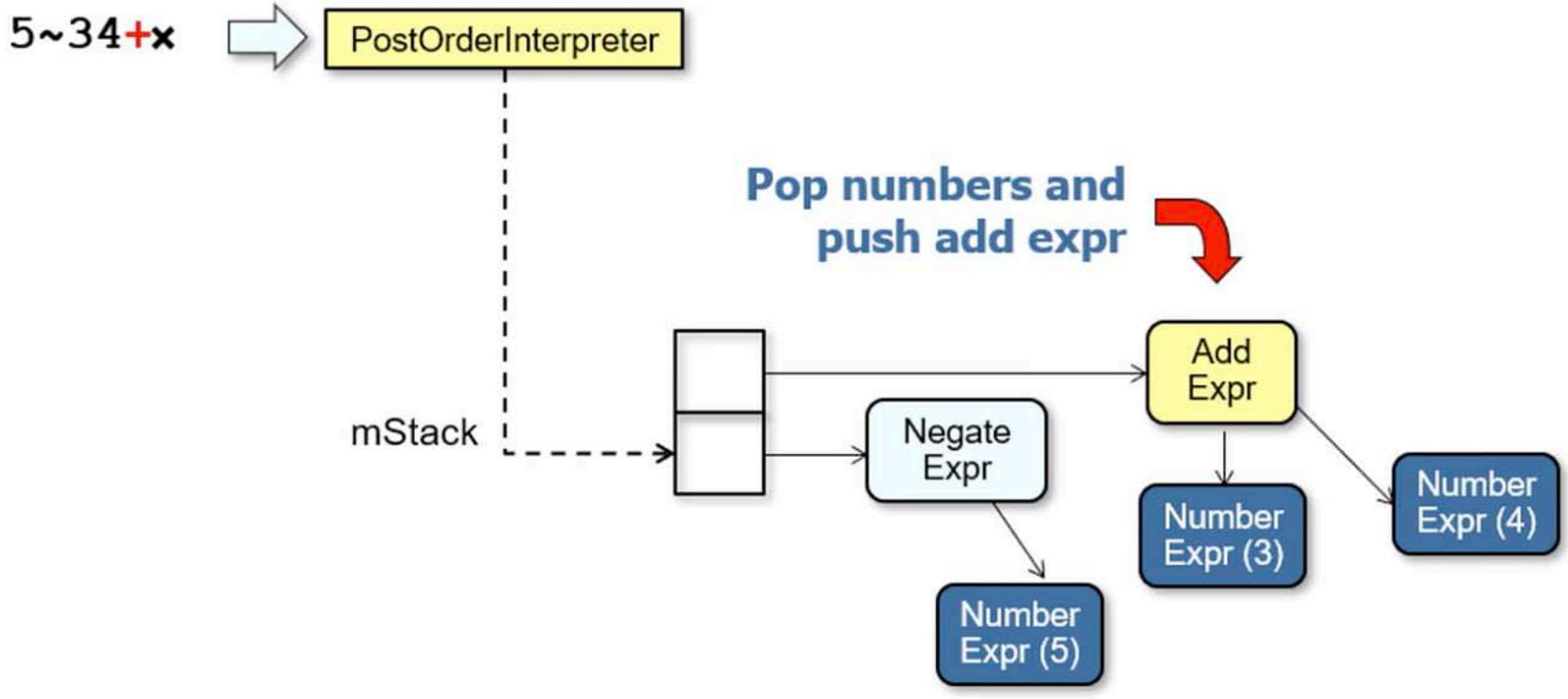
Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.



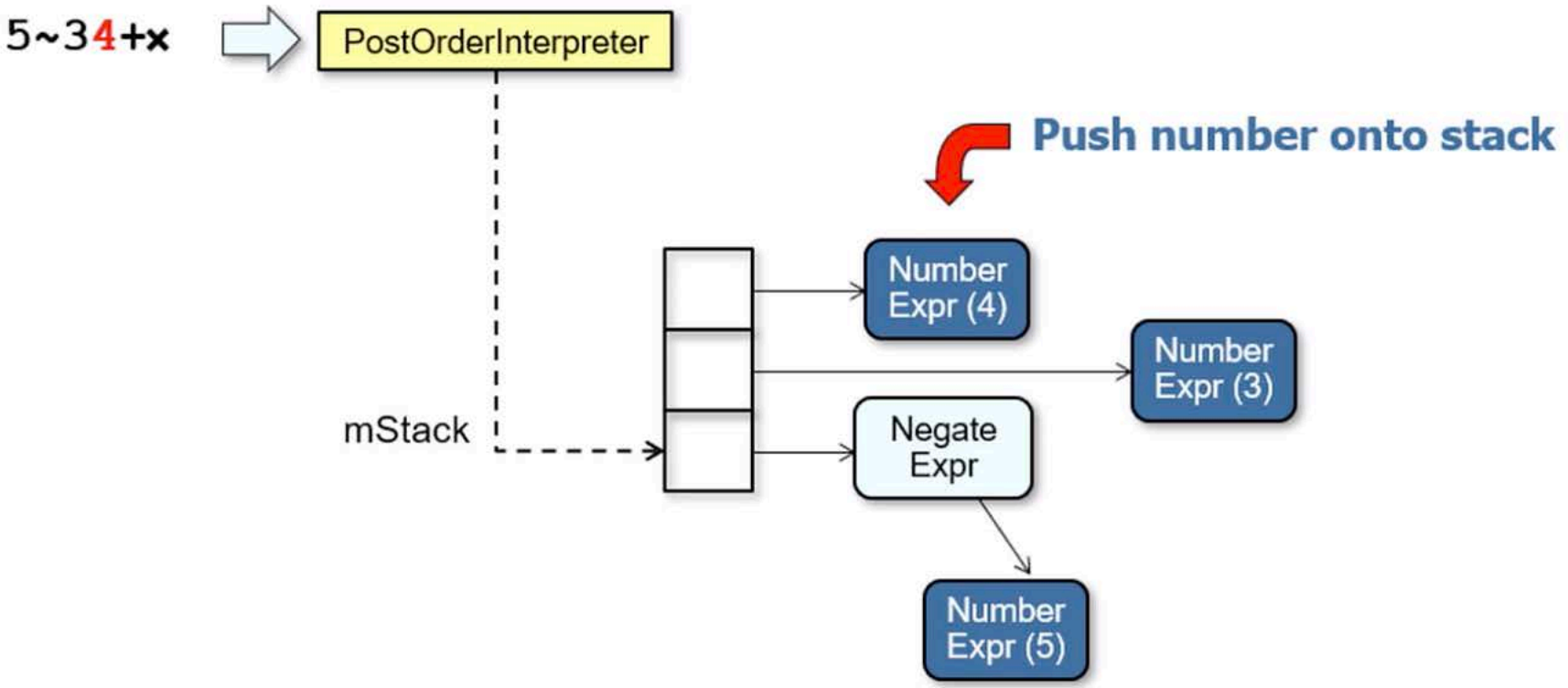
Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.



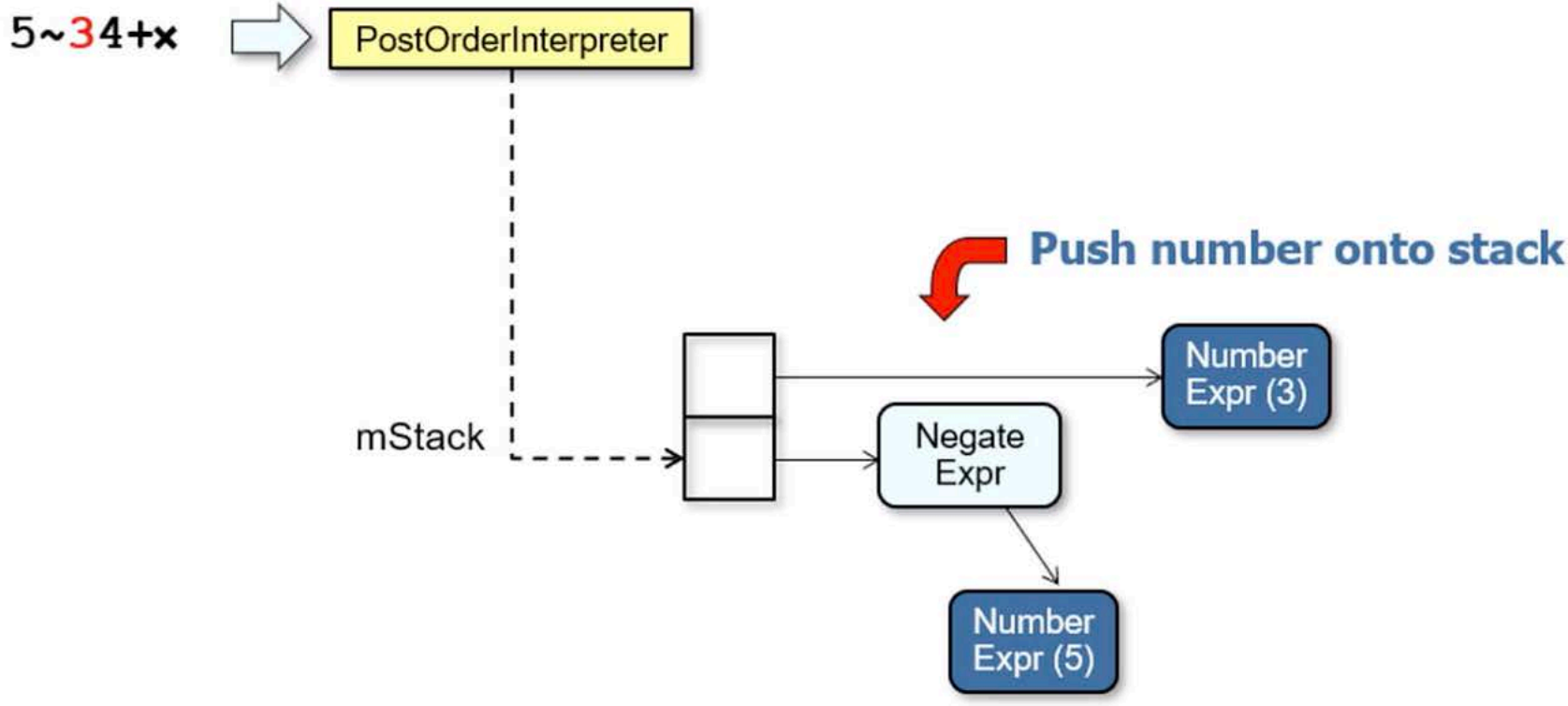
Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.



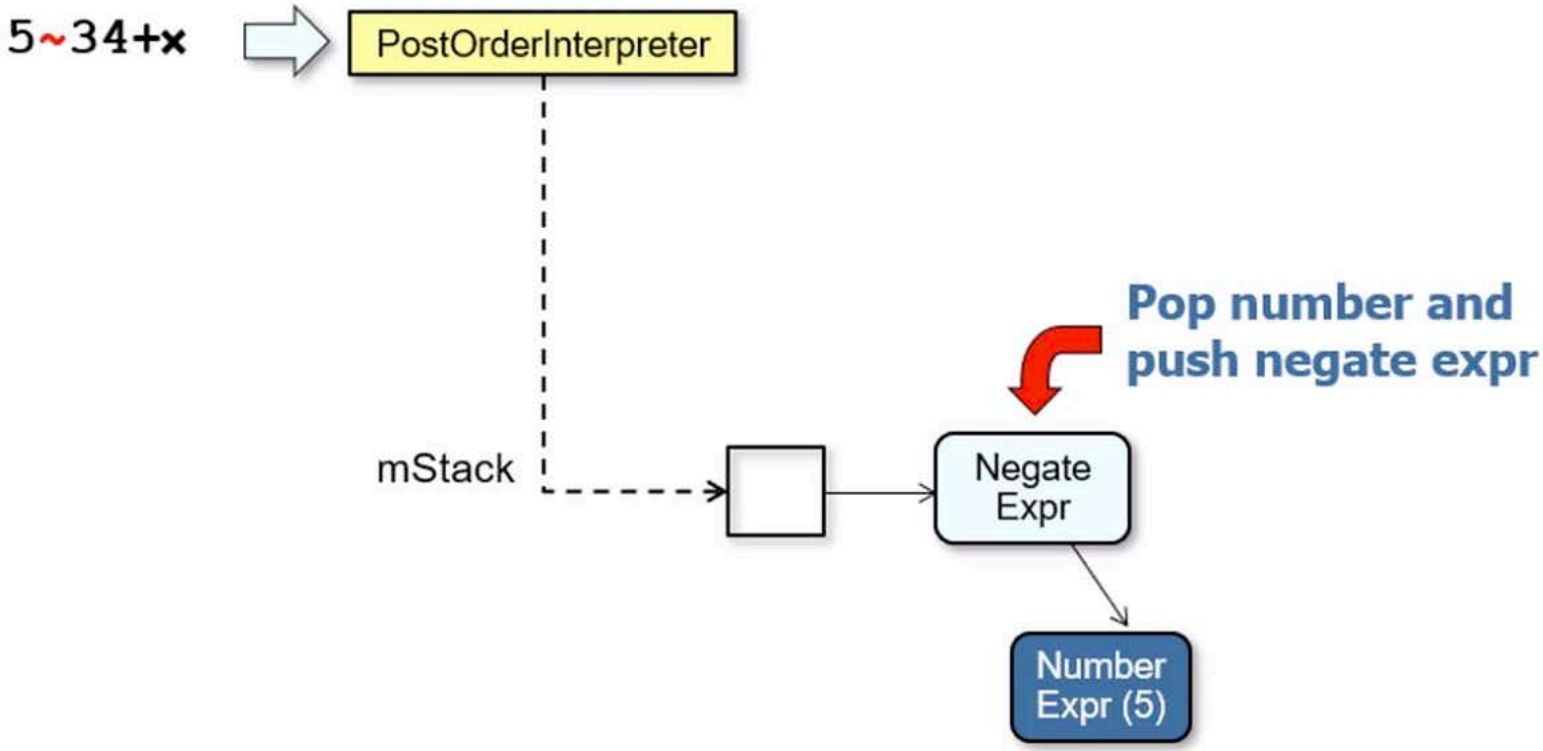
Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.



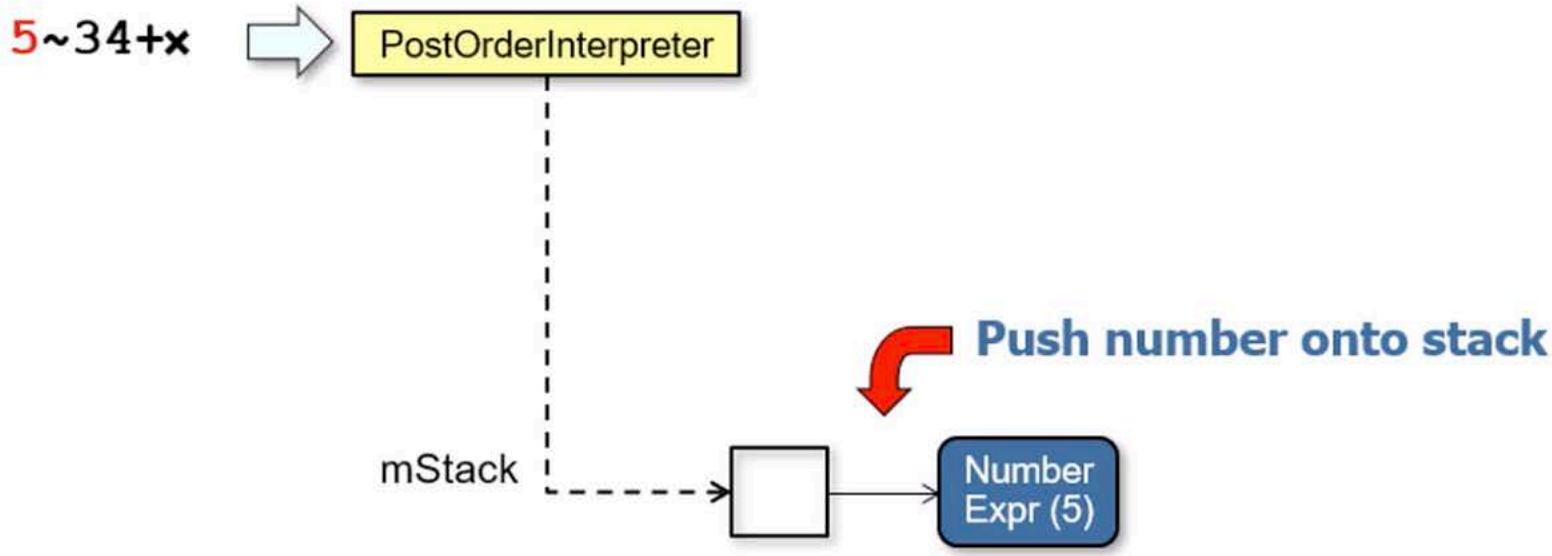
Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.



Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.



Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.

```
class ExprSpliterator
      extends Spliterators.AbstractSpliterator<Expr> {
  ...
  public boolean tryAdvance(Consumer<? super Expr> unUsed) {
    ...
    else { ...
      else {
        Expr rightExpr = mStack.pop();
        switch (c) {
          case '+':
             mStack.push(new AddExpr(mStack.pop(), rightExpr));
             break;
          case '-':
             mStack.push(new SubtractExpr(mStack.pop(),
                                          rightExpr));
             break;
          ...
```

Pop the top operand off the stack

See www.geeksforgeeks.org/stack-set-4-evaluation-postfix-expression

Interpreter example in Java

- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression.

```
class ExprSpliterator
      extends Spliterators.AbstractSpliterator<Expr> {
  ...
  public boolean tryAdvance(Consumer<? super Expr> unUsed) {
    ...
    else { ...
      else {
        Expr rightExpr = mStack.pop();
        switch (c) {
          case '+':
             mStack.push(new AddExpr(mStack.pop(), rightExpr));
             break;
          case '-':
             mStack.push(new SubtractExpr(mStack.pop(),
                                          rightExpr));
             break;
          ...
```

Handle the addition operator

The Interpreter Pattern

Other Considerations

Douglas C. Schmidt

Consequences

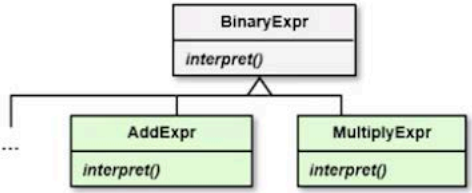
+ Simple grammars are easy to change and extend
  - E.g., all rules represented by distinct classes in a consistent and orderly manner

```
expr ::= factor expr-tail

expr-tail ::= add_sub expr
            | /* empty */;

factor ::= term factor-tail

factor-tail ::= mul_div factor
              | /* empty */;

mul_div ::= '*' | '/'

add_sub ::= '+' | '-'

term :: NUMBER | '(' expr ')'
```

This grammar removes immediate left recursion.

Consequences

+ Simple grammars are easy to change and extend
+ Adding another rule adds another class




```
expr ::= factor expr-tail

expr-tail ::= add_sub expr
            | /* empty */;

factor ::= term factor-tail

factor-tail ::= mul_div factor
              | /* empty */;

mul_div ::= '×' | '/'

add_sub ::= '+' | '-'

term :: NUMBER | '(' expr ')'
```

Consequences

- Complex grammars are hard to create and maintain
  - E.g., more rules that are interdependent yield more interdependent classes

```
postfix-expression ::=
primary-expression
postfix-expression [ expression ]
postfix-expression ( expression-list_opt )
simple-type-specifier ( expression-list_opt )
typename::_opt nested-name-
   specifier identifier ( expression-list_opt )
typename::_opt nested-name-specifier template_opt
   template-id ( expression-list_opt )
postfix-expression . template_opt id-expression
postfix-expression -> template_opt id-expression
postfix-expression . pseudo-destructor-name
postfix-expression -> pseudo-destructor-name
postfix-expression ++
postfix-expression --
dynamic_cast < type-id > ( expression )
static_cast < type-id > ( expression )
reinterpret_cast < type-id > ( expression )
const_cast < type-id > ( expression )
type-id ( expression )
type-id ( type-id )
expression-list ::=
assignment-expression
expression-list , assignment-expression
```

Consequences

– Complex grammars are hard to create and maintain
  - E.g., more rules that are interdependent yield more interdependent classes

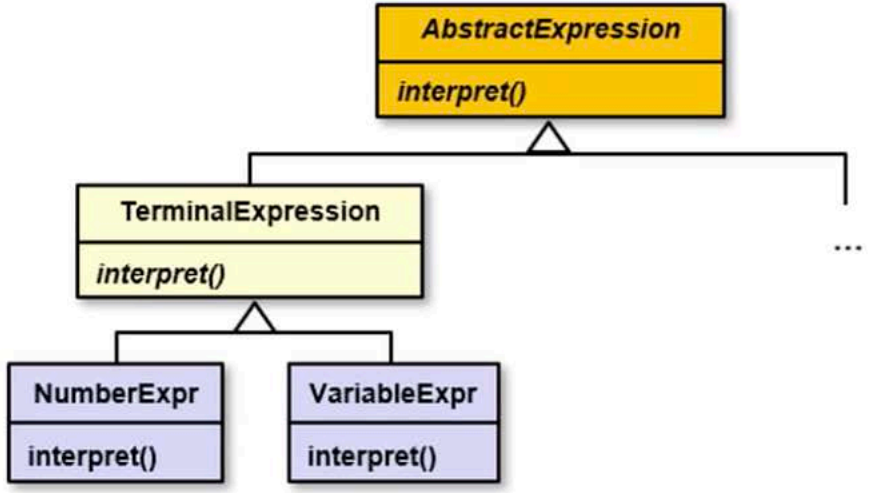
Complex grammars often require different approach, e.g., parser generators.

JavaCC™

See www.antlr.org and javacc.org

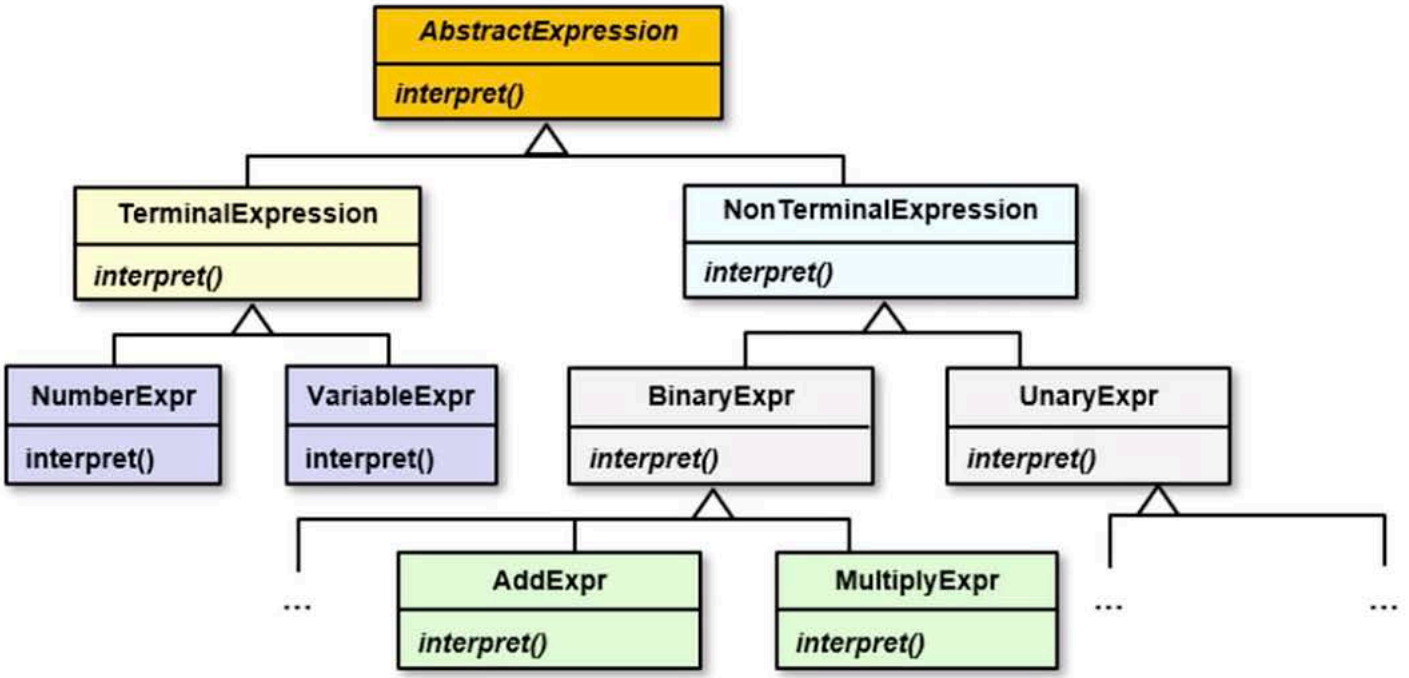
Implementation

- Various approaches:
  - Express language rules, one per class
    - Literal translations expressed as *terminal expressions*



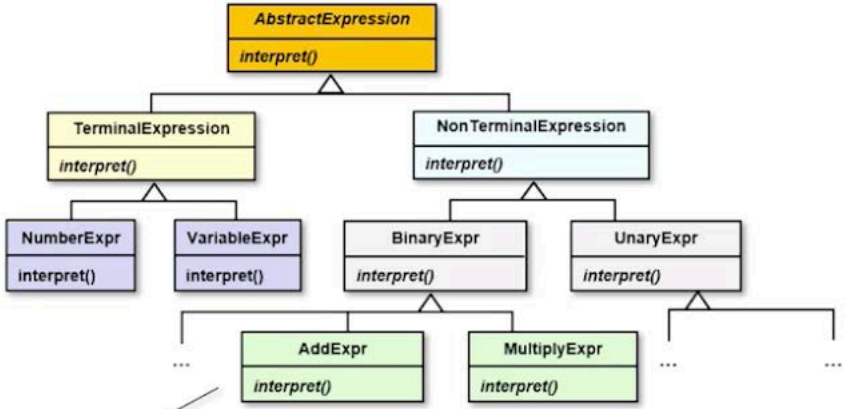
Implementation

- Various approaches:
  - Express language rules, one per class.
    - Literal translations expressed as *terminal expressions*
    - Binary and unary nodes expressed as *nonterminal expressions*



Implementation

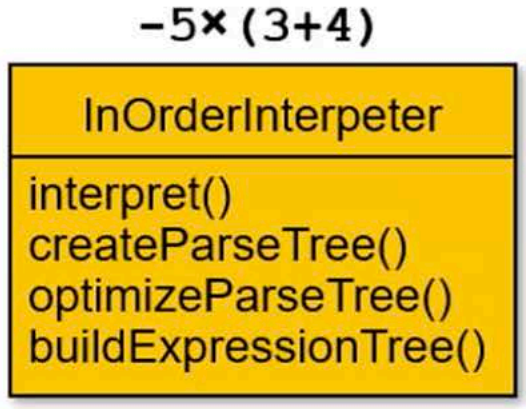
- Various approaches:
  - Express language rules, one per class.
    - Literal translations expressed as *terminal expressions*
    - Binary and unary nodes expressed as *nonterminal expressions*



This approach can yield a large number of classes, but they mimic the grammar

Implementation

- Various approaches:
  - Express language rules, one per class.
  - Use operator precedence for an interpreter that builds parse trees from in-order expressions.

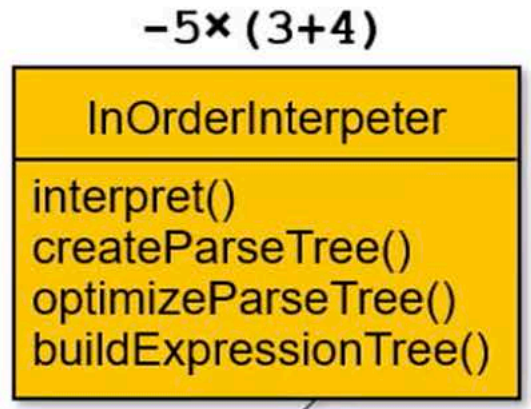



```
public final static int sMULTIPLICATION = 0;
public final static int sDIVISION = 1;
public final static int sADDITION = 2;
public final static int sSUBTRACTION = 3;
public final static int sNEGATION = 4;
public final static int sLPAREN = 5;
public final static int sRPAREN = 6;
public final static int sID = 7;
public final static int sNUMBER = 8;
public final static int sDELIMITER = 9;

public final static int mTopOfStackPrecedence[] = {
    12, 11, 7, 6, 10, 2, 3, 15, 14, 1
};

public final static int mCurrentTokenPrecedence[] = {
    9, 8, 5, 4, 13, 18, 2, 17, 16, 1
};
```

See en.wikipedia.org/wiki/Operator-precedence_parser

Implementation

- Various approaches:
  - Express language rules, one per class.
  - Use operator precedence for an interpreter that builds parse trees from in-order expressions.



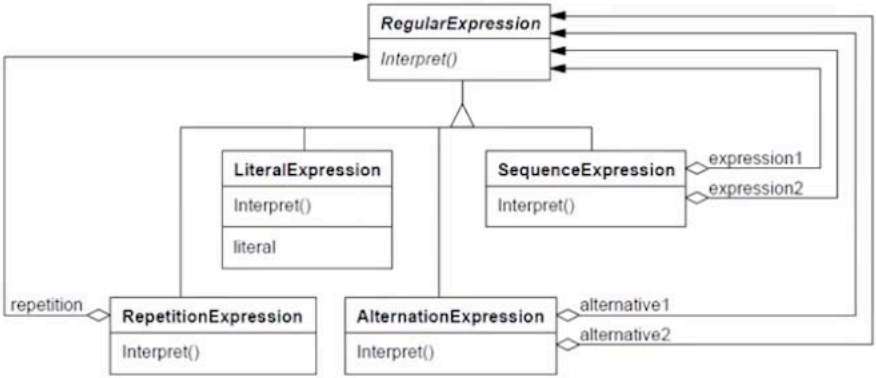
This approach has fewer classes, but the software design does not mimic the grammar.

```
public final static int sMULTIPLICATION = 0;
public final static int sDIVISION = 1;
public final static int sADDITION = 2;
public final static int sSUBTRACTION = 3;
public final static int sNEGATION = 4;
public final static int sLPAREN = 5;
public final static int sRPAREN = 6;
public final static int sID = 7;
public final static int sNUMBER = 8;
public final static int sDELIMITER = 9;

public final static int mTopOfStackPrecedence[] = {
    12, 11, 7, 6, 10, 2, 3, 15, 14, 1
};

public final static int mCurrentTokenPrecedence[] = {
    9, 8, 5, 4, 13, 18, 2, 17, 16, 1
};
```

See ExpressionTree/CommandLine/src/expressiontree/interpreter/inorder

Known uses

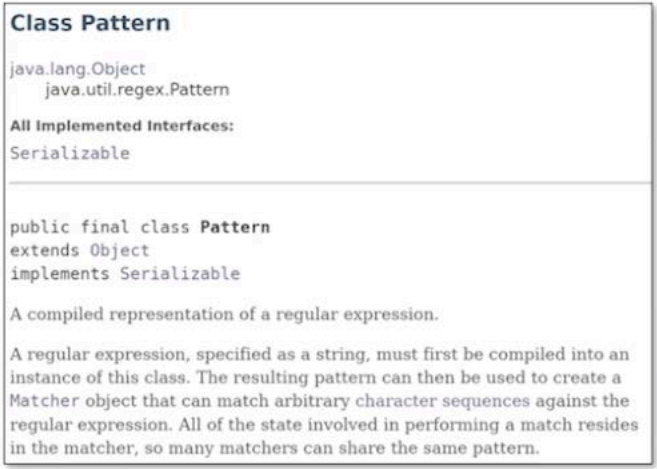
- Text editors and web browsers use the *Interpreter* pattern to lay out documents and check spelling
  - E.g., an equation in TeX is represented as a tree where internal nodes are operators and leaves are variables
- Smalltalk compilers
- Regular expression parsers



See www.drdobbs.com/a-different-interpretation-of-the-interp/184401605

Known uses

- Text editors and web browsers use the *Interpreter* pattern to lay out documents and check spelling
  - E.g., an equation in TeX is represented as a tree where internal nodes are operators and leaves are variables
- Smalltalk compilers
- Regular expression parsers
  - E.g., java.util.regex.Pattern



Class Pattern

java.lang.Object
    java.util.regex.Pattern

All Implemented Interfaces:

Serializable

```
public final class Pattern
extends Object
implements Serializable
```

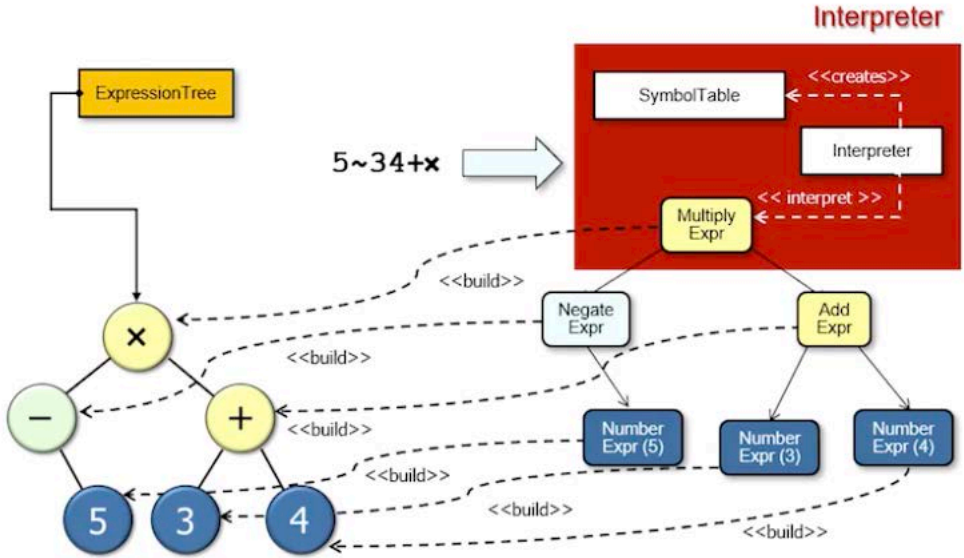
A compiled representation of a regular expression.

A regular expression, specified as a string, must first be compiled into an instance of this class. The resulting pattern can then be used to create a Matcher object that can match arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the matcher, so many matchers can share the same pattern.

See docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html

Summary of the Interpreter Pattern

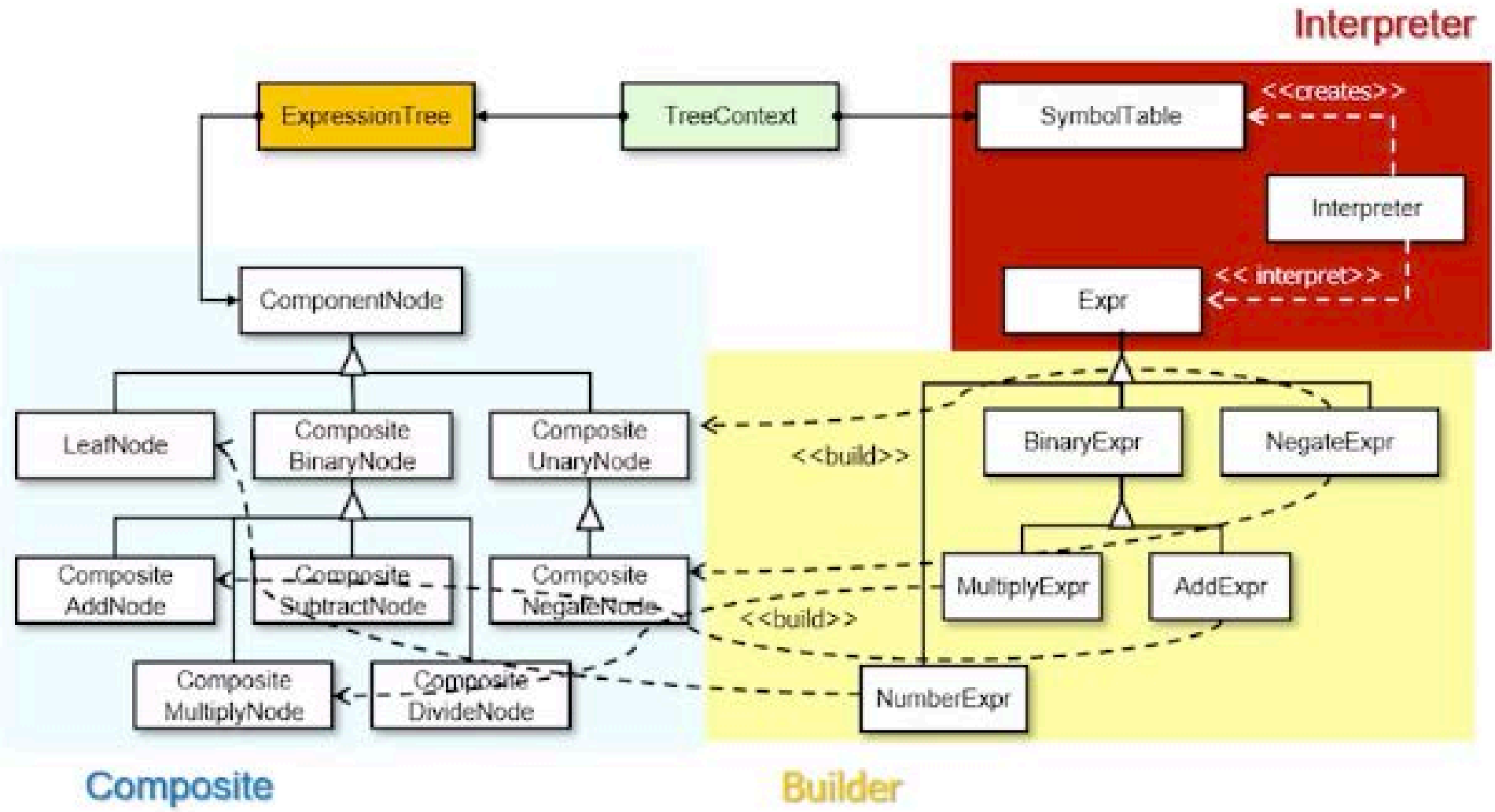
- *Interpreter* automatically converts a user input expression into a parse tree, which is then used to build the corresponding expression tree.



Next, we cover a *Creational* pattern for building an expression tree from a parse tree.

A Pattern for Building Objects Incrementally

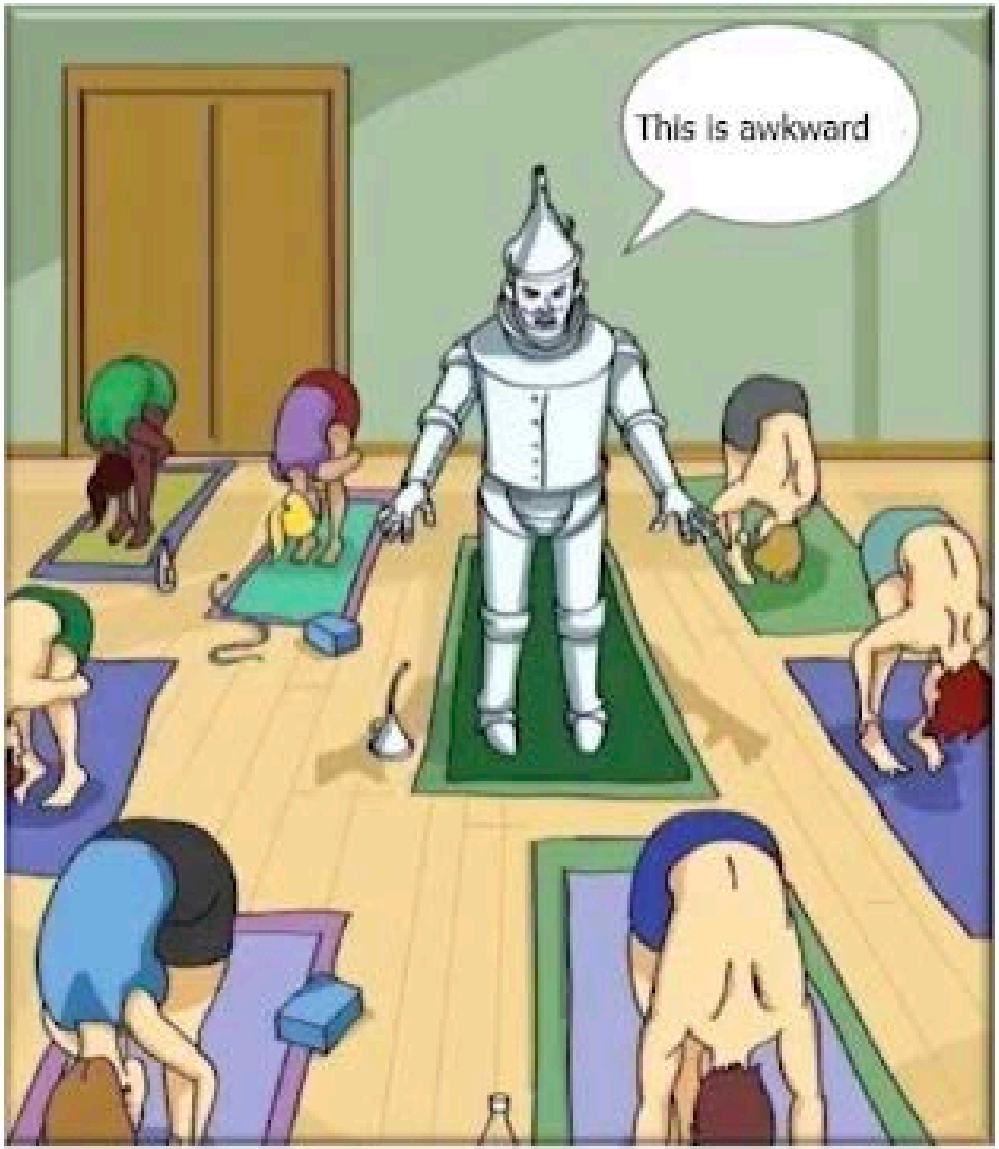
***Purpose**: Recursively builds the expression tree's Composite-based internal data structure from the Interpreter-generated parse tree*

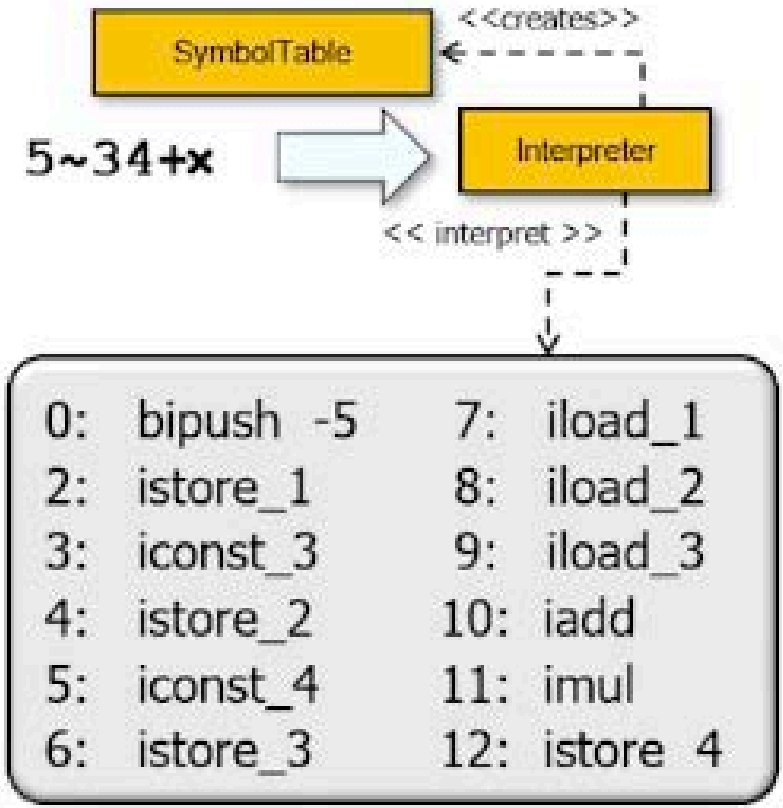


There are *many* classes in this design, but only a handful of patterns.

Problem: Inflexible Interpreter Output

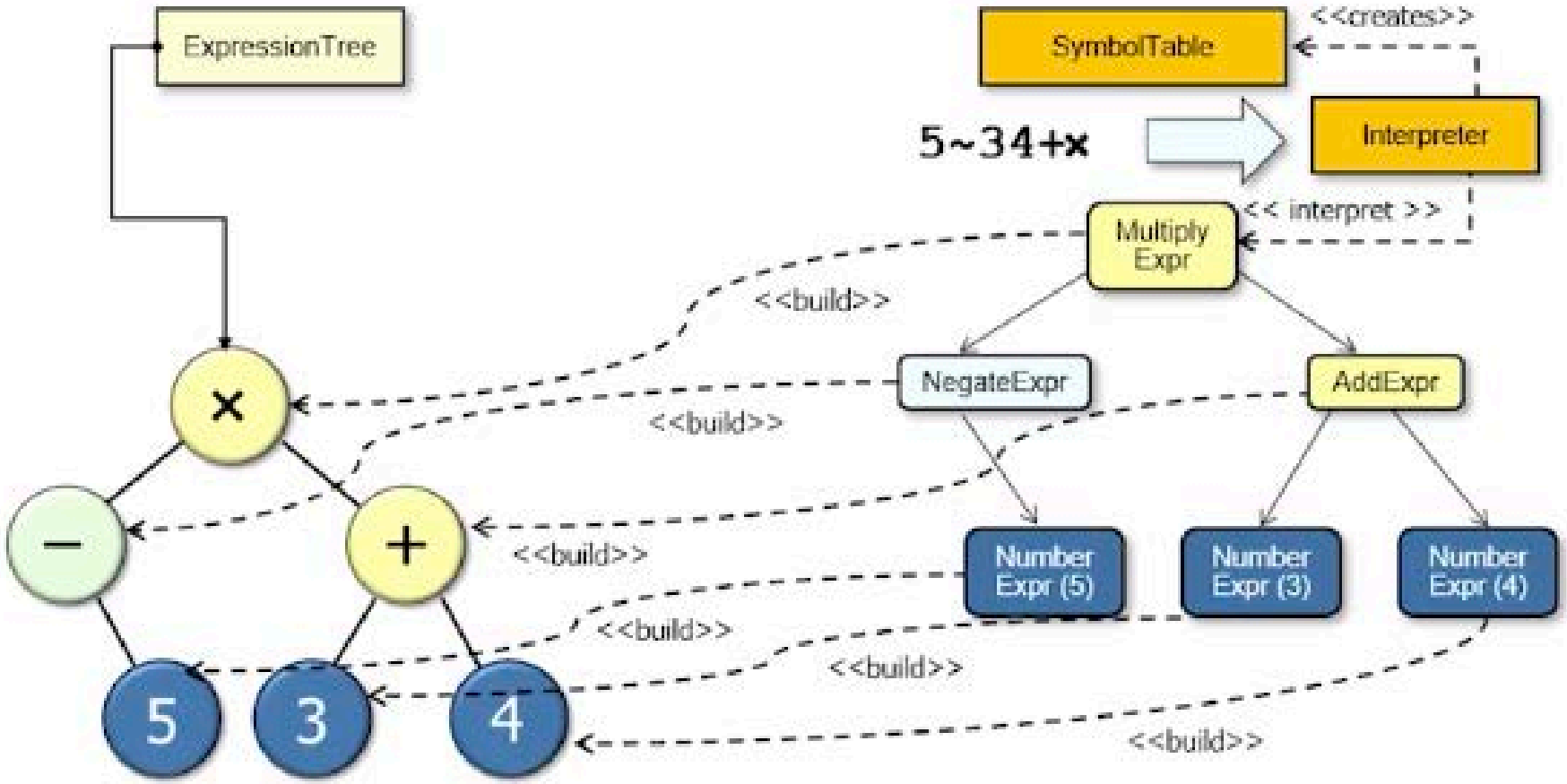
- Hard-coding `Interpreter` to only generate one type of output is inflexible.





Solution: Build Complex Object Incrementally

- Traverse the resulting parse tree recursively to build the nodes in the corresponding expression tree composite.



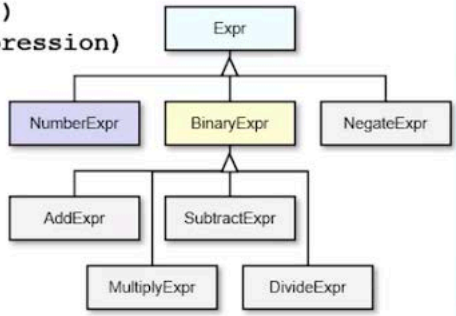
The expression tree representation may be quite different from the parse tree.

Expr Class Hierarchy Overview

- Represents the nodes in the parse tree, which are used to build the corresponding nodes in the expression tree

Class methods

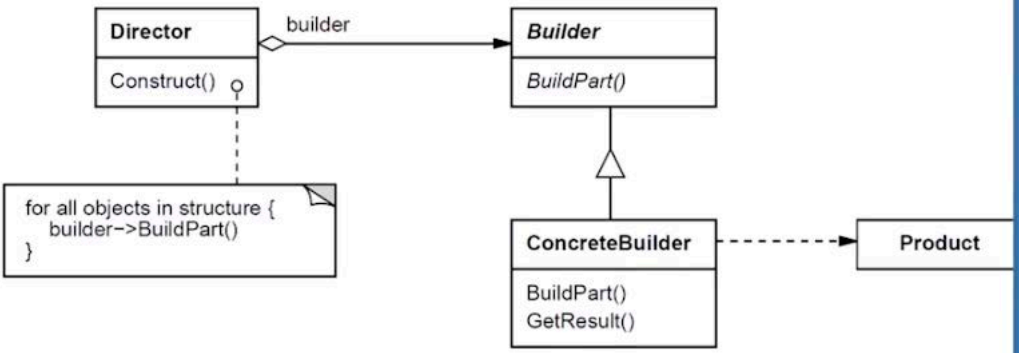
```
          void optimizeParseTree()
ExpressionTree buildExpressionTree()
ExpressionTree interpret(String expression)
```



- **Commonality**: provides a common interface building parse trees and expression trees from user input expressions
- **Variability**: the structure of the parse trees and expression trees can vary depending on the format, contents, and optimization of input expressions

Learning Objectives in This Lesson

- Recognize how the *Builder* pattern can be applied to incrementally build an expression tree from a parse tree.
- Understand the structure and functionality of the *Builder* pattern.

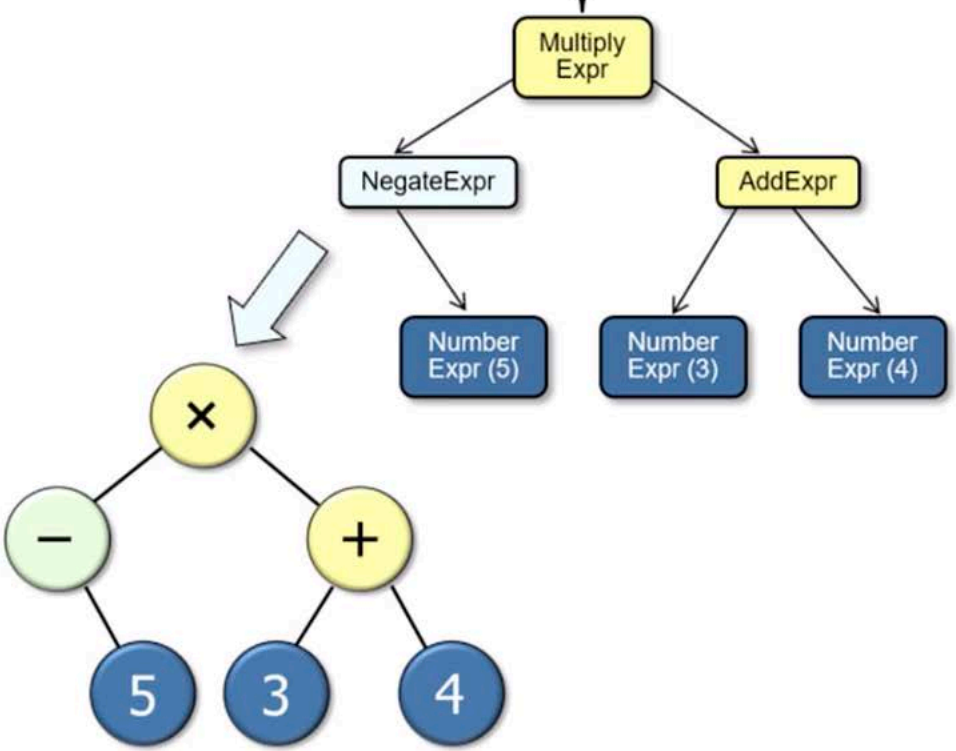


Builder GoF Object Creational

Intent

- Separate construction of a complex object from its representation

"Pre-order" input = `×-5+34`
"Post-order" input = `5-34+×`
"Level-order" input = `×-+534`
"In-order" input = `-5×(3+4)`

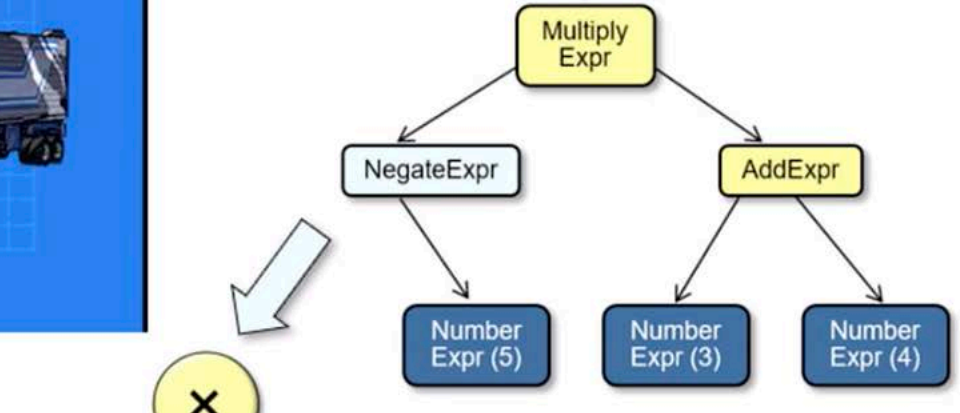


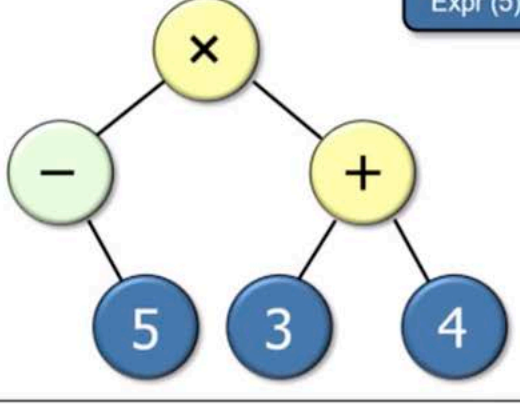
See en.wikipedia.org/wiki/Builder_pattern

Applicability

- Need to isolate the knowledge of creating a complex object from its parts

"Pre-order" input = ×-5+34
"Post-order" input = 5-34+×
"Level-order" input = ×-+534
"In-order" input = -5×(3+4)

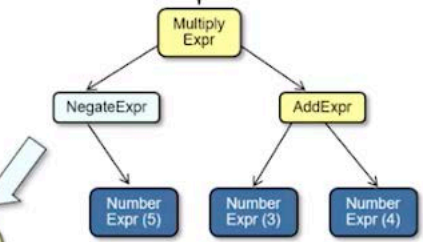


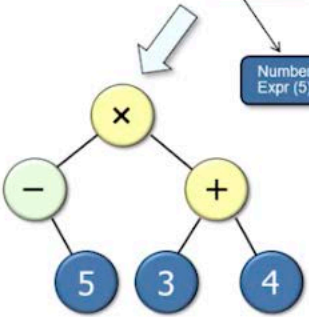


Applicability

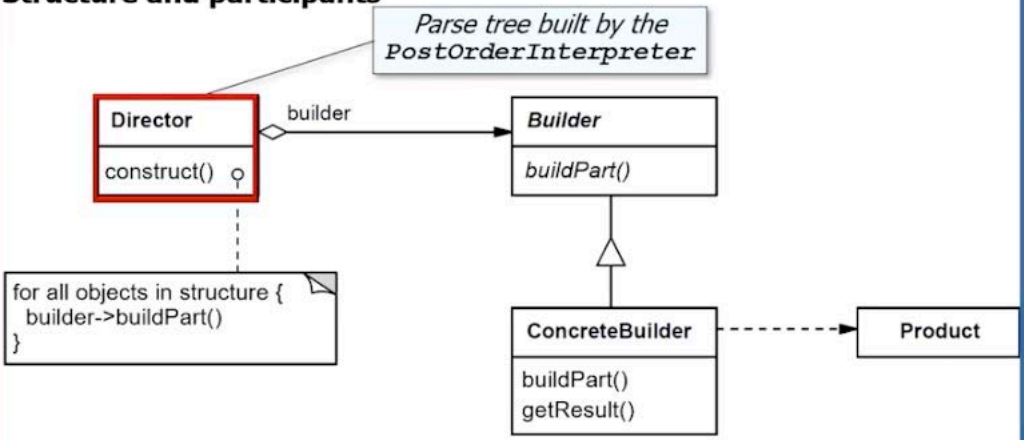
- Need to isolate the knowledge of creating a complex object from its parts
- Need to allow different implementations and (internal) interfaces of an object's

"Pre-order" input = ×−5+34
"Post-order" input = 5−34+×
"Level-order" input = ×−+534
"In-order" input = −5×(3+4)

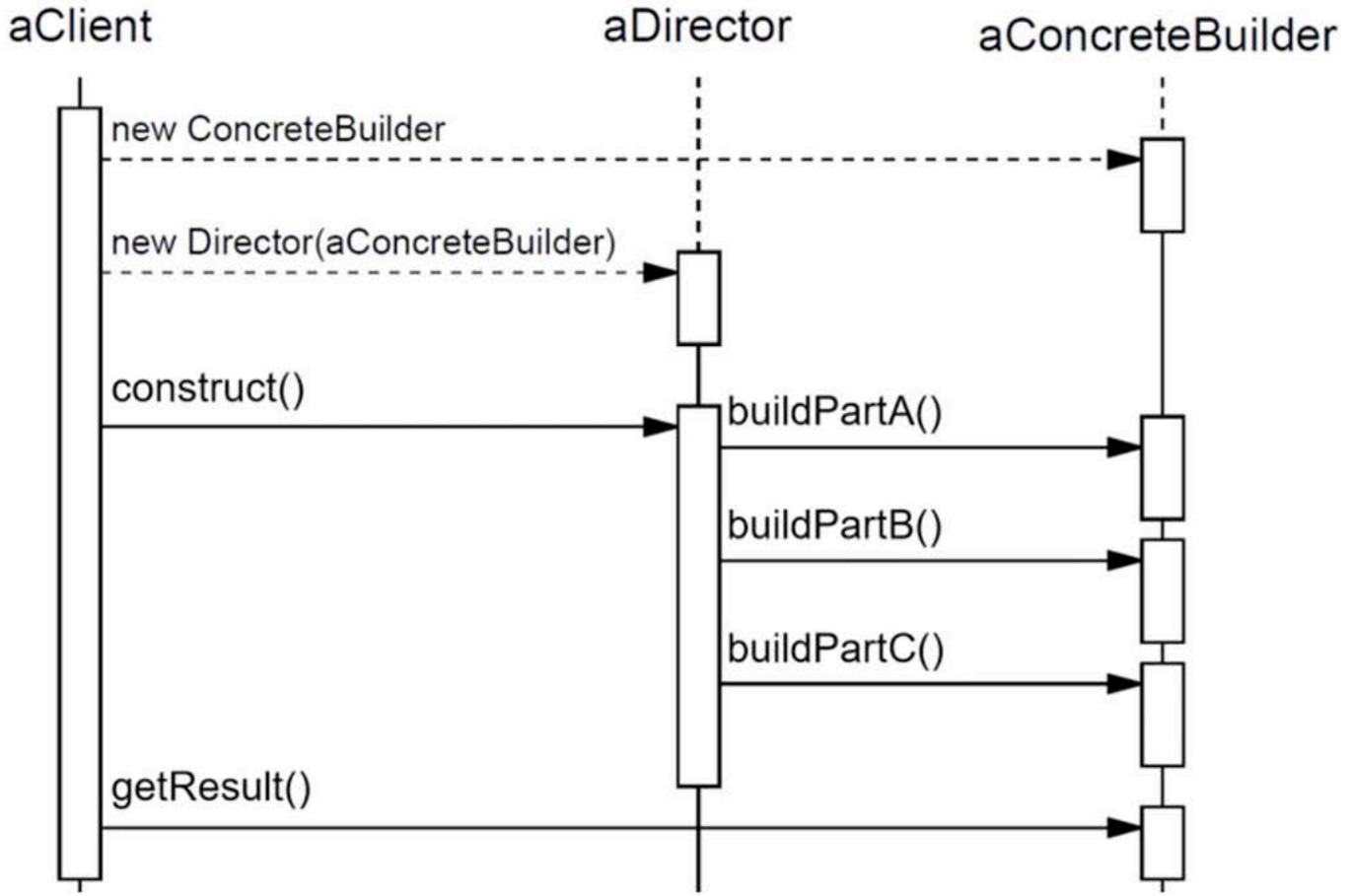




Structure and participants



Collaborations

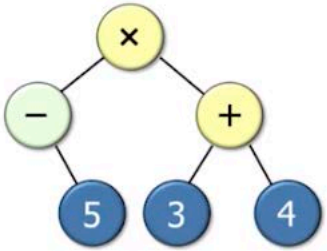


Consequences

+ Isolates the code for construction and representation
+ Finer control over the construction process
+ Can vary a product's internal representation

```
class NumberExpr extends Expr {
  ComponentNode build() {
    return new TreeNode(item);
  }
  ...
}
```

```
class AddExpr extends BinaryExpr {
  ComponentNode build() {
    return new TreeNode('+',
     (left.build(), right.build());
  ...
}
```



Consequences

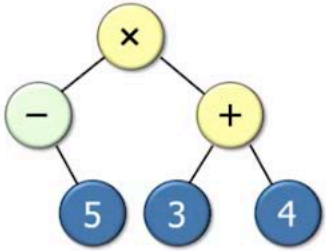
+ Isolates the code for construction and representation

+ Finer control over the construction process

Every composite node controls how it's constructed.

```
class NumberExpr extends Expr {
  ComponentNode build() {
    return new LeafNode(mItem);
  }
  ...
}
```

```
class AddExpr extends BinaryExpr {
  ComponentNode build() {
    return new CompositeAddNode
     (mLeftExpr.build(),
      mRightExpr.build());
  ...
}
```

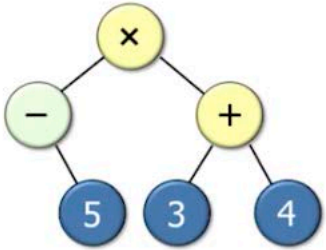


Consequences

+ Isolates the code for construction and representation

```
class NumberExpr extends Expr {
  ComponentNode build() {
    return new LeafNode(mItem);
  }
  ...
}
```

```
class AddExpr extends BinaryExpr {
  ComponentNode build() {
    return new CompositeAddNode
     (mLeftExpr.build(),
      mRightExpr.build());
  ...
}
```



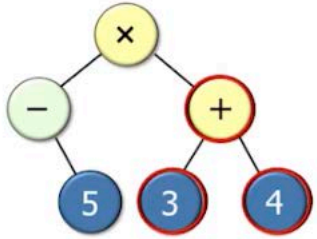
The Builder Pattern

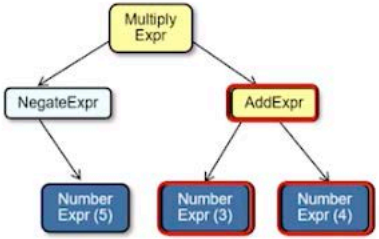
Other Considerations

Douglas C. Schmidt

Builder example in Java

- `buildExpressionTree()` builds a composite expression tree from a parse tree.





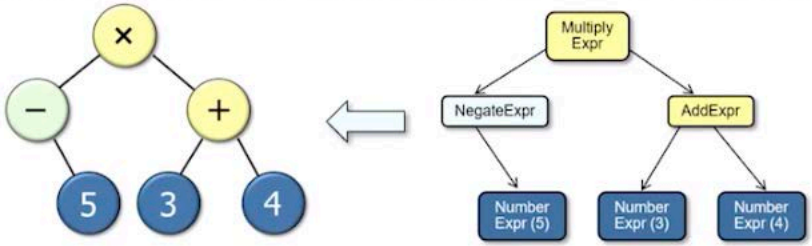

```
class AddExpr
      extends BinaryExpr {
  ComponentNode build() {
    return new
      CompositeAddNode
      (mLeftExpr.build(),
       mRightExpr.build());
  }
  ...
```

Build component nodes recursively

```
class NumberExpr
      extends Expr {
  ComponentNode build() {
    return new LeafNode(mItem);
  }
  ...
```

Builder example in Java

- `buildExpressionTree()` builds a composite expression tree from a parse tree.

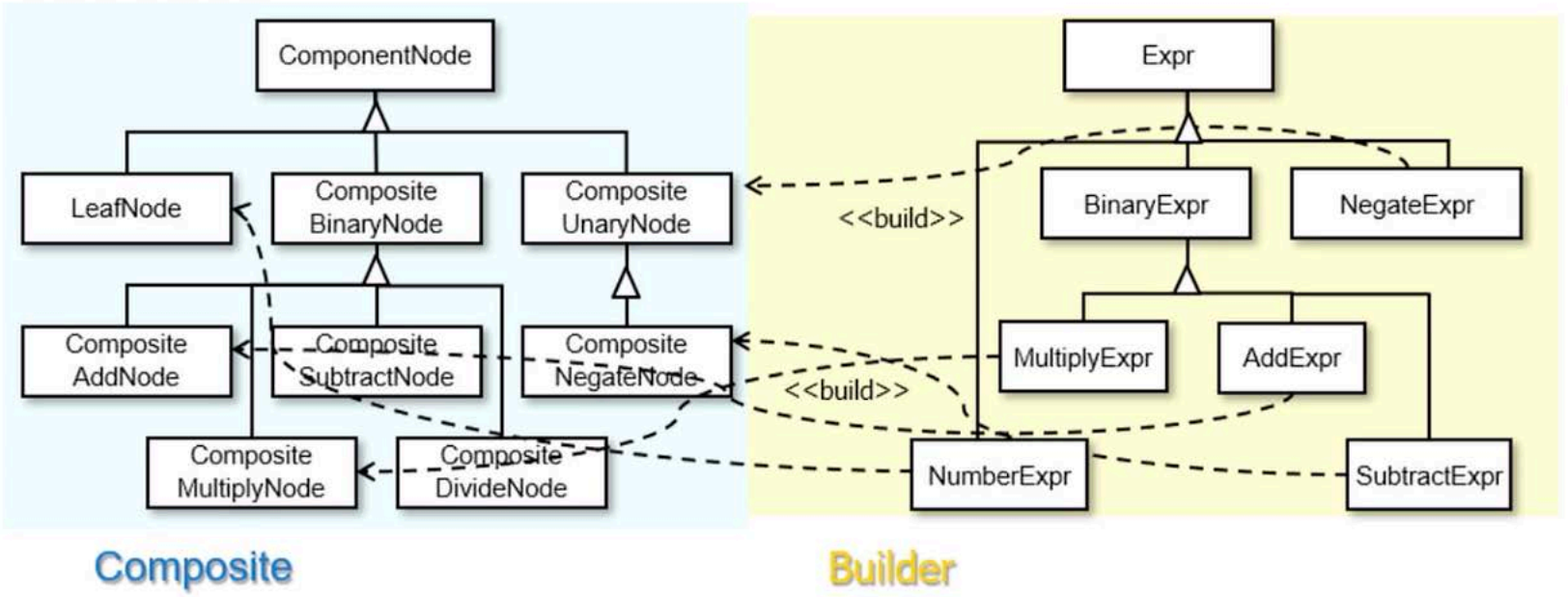



```
class PostOrderInterpreter extends InterpreterImpl {
  protected ExpressionTree buildExpressionTree(Expr parseTree) {
    return expressionTreeFactory
                        .makeExpressionTree(parseTree.build());
  }
```

See ExpressionTree/CommandLine/src/expressiontree/interpreter/postorder

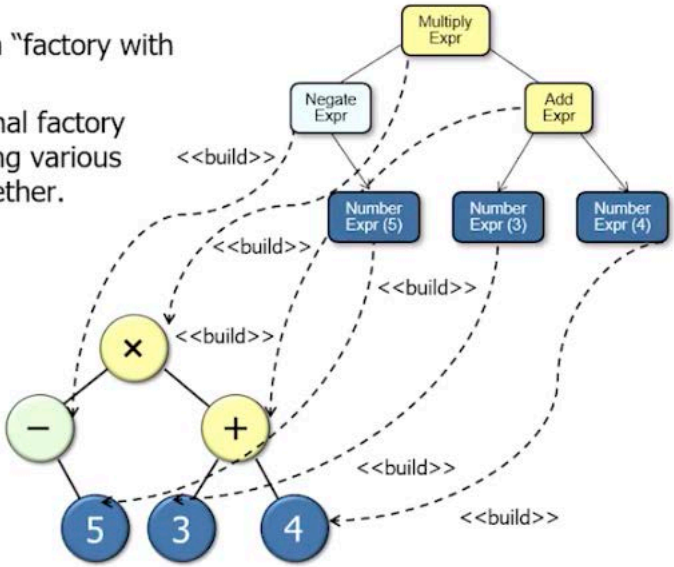
Consequences

– May involve a lot of classes and class interdependencies



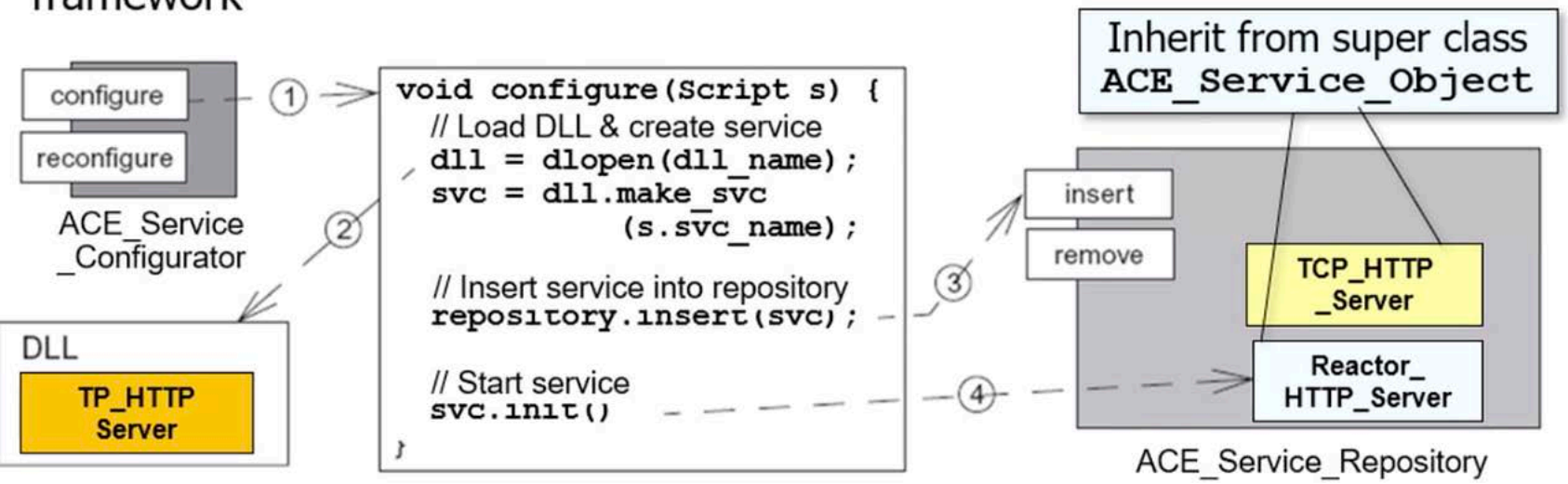
Implementation

- The Builder pattern is a "factory with a mission."
  - It extends conventional factory patterns by connecting various implementations together.



Known uses

- ET++ RTF converter
- Smalltalk-80
- ACE Service Configurator framework



Builder

Known uses

- ET++ RTF converter
- Smalltalk-80
- ACE Service Configurator framework
- "Effective Java" style

Builds a new object

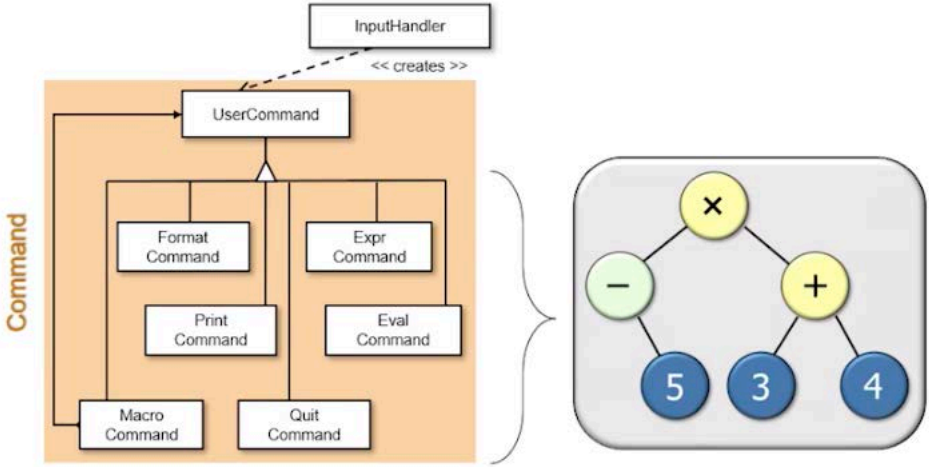
```
class Builder {
  String mSum = ""; String mDesc = "";
  ...
  public Builder() { /* default ctor */ }

  private Builder(String sum, String desc)
  {mSum = sum; mDesc = desc; }

  public Builder setSummary(String sum)
  { mSum = sum; return this; }

  public Builder setDesc(String desc)
  { mDesc = desc; return this; }

  public Builder build()
  { return new Builder(mSum, mDesc); }
  ...
```

```
public class Test {
  public static void
  main(String[] a) {
    Builder task =
      new Builder()
      .setDesc("Builder test").setSummary("Cool!").build();
    ...
  } ...
```

See www.vogella.com/tutorials/DesignPatternBuilder/article.html

A Pattern for Objectifying User Requests

***Purpose**: Define objectified actions that enable users to perform command requests consistently and extensibly in the expression tree processing app.*



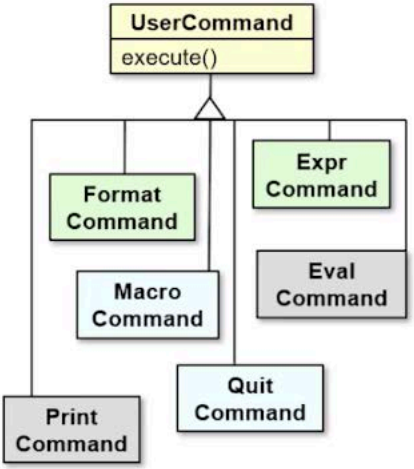
Command provides a uniform means to process all user-requested commands.

Problem: Scattered/Fixed User Request Implementations

- Hard-coding the program to handle only a fixed set of user commands impedes the evolution that's needed to support new requirements.

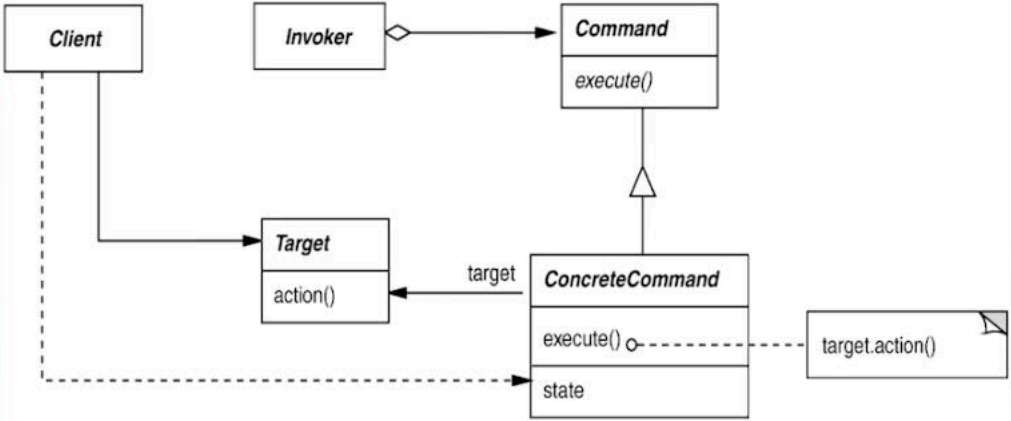
Solution: Encapsulate User Requests as Commands

- Create a hierarchy of `UserCommand` subclasses



Learning Objectives in This Lesson

- Recognize how the *Command* pattern can be applied to perform user-requested commands consistently and extensibly in the expression tree processing app.
- Understand the structure and functionality of the *Command* pattern.



The Builder Pattern

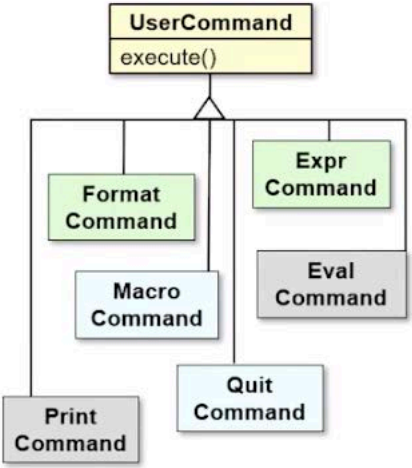
Structure and Functionality

Douglas C. Schmidt

Command GoF Object Behavioral

Intent

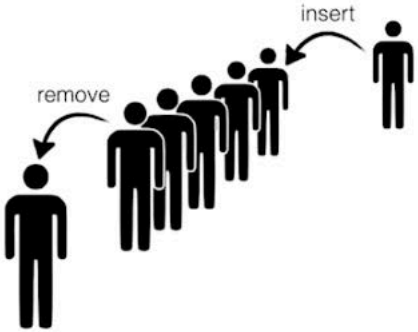
- Encapsulate the request for a service as an object

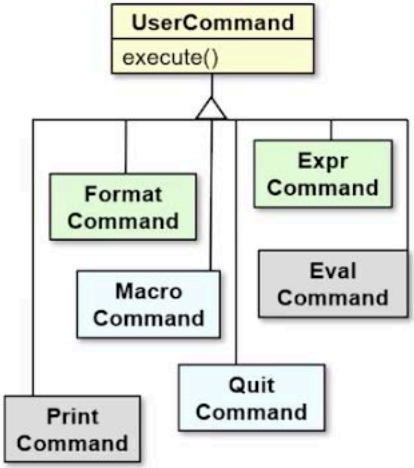


See en.wikipedia.org/wiki/Command_pattern

Applicability

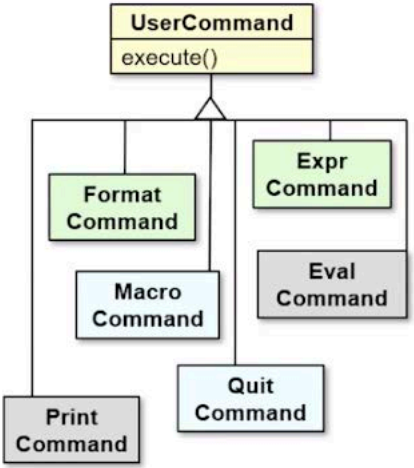
- Want to parameterize objects with an action to perform
- Want to specify, queue, and execute requests at different times



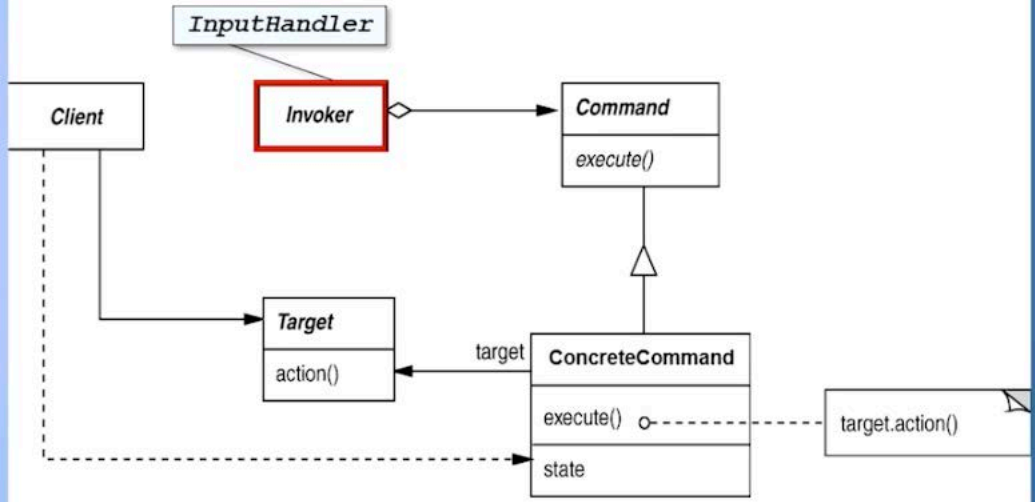


Applicability

- Want to parameterize objects with an action to perform
- Want to specify, queue, and execute requests at different times
- Want to support multilevel undo/redo



Structure and participants



The Command Pattern

Implementation in Java

Douglas C. Schmidt

Command example in Java

- Encapsulate the execution of a sequence of commands as an object, which is used to implement the "succinct mode."

```
public class MacroCommand extends UserCommand {
  ...
  private List<UserCommand> mMacroCommands = new ArrayList<>();



  MacroCommand(TreeContext context,
               List<UserCommand> macroCommands) {
    super(context); mMacroCommands = macroCommands;
  }

  public void execute() throws Exception {
    mMacroCommands.forEach(UserCommand::execute);
  }
  ...
```

The Java 8 way of executing a sequence of commands to implement the "succinct mode"

See blog.amitinside.com/Command-Pattern-Using-Java-8-Lambda

The Command Pattern

Other Considerations

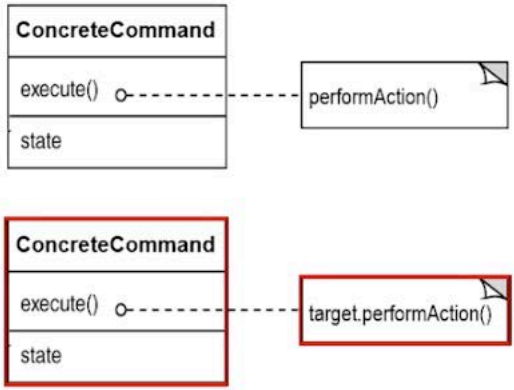
Douglas C. Schmidt

Consequences

+ Abstracts the executor of a service
  - Makes programs more modular and flexible

Consequences

+ Abstracts the executor of a service
  - Makes programs more modular and flexible, e.g.,
    - Can bundle state and behavior into an object
    - Can forward behavior to other objects



See upcoming lesson on the *State* pattern for an example of forwarding.

Consequences

+ Abstracts the executor of a service
  - Makes programs more modular and flexible, e.g.,
    - Can bundle state and behavior into an object
    - Can forward behavior to other objects
    - Can extend behavior via subclassing
    - Can pass a command object as a parameter

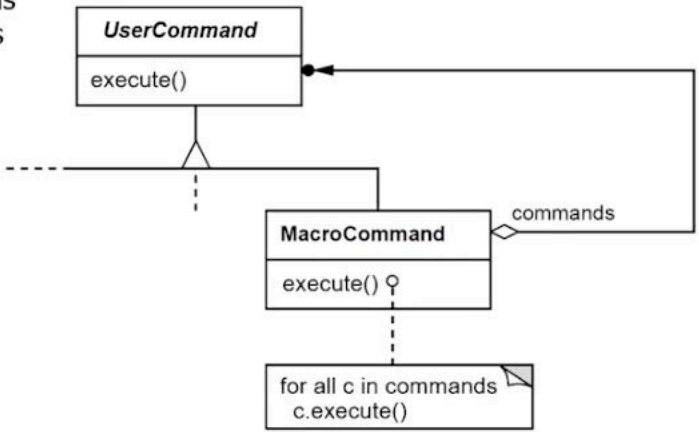
```
void handleInput() {
  ...
  UserCommand command =
    makeUserCommand(input);



  executeCommand(command);
```

The handleInput() method in InputHandler plays the role of "invoker."

Consequences

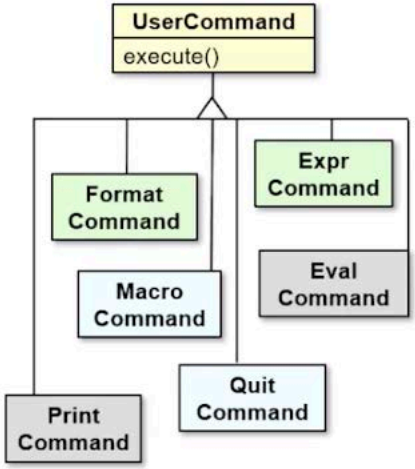
+ Abstracts the executor of a service

+ Composition yields macro commands



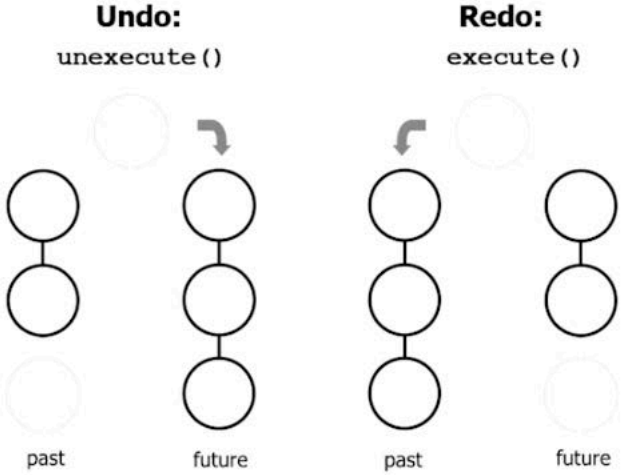
Consequences

- Might result in lots of trivial command subclasses



Consequences

- Might result in lots of trivial command subclasses
- Excessive memory may be needed to support undo/redo operations



Command GoF Object Behavioral

Known uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- GNU Emacs
- Microsoft Office tools
- Java **`Runnable`** interface
  - **`Runnable`** can also be used to implement the *Command Processor* pattern

java.lang

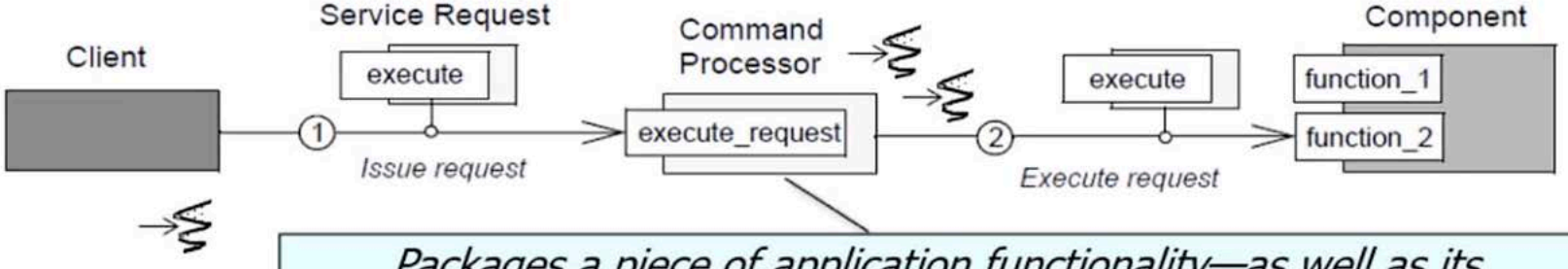
Interface Runnable

All Known Subinterfaces:
RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:
AsyncBoxView.ChildState, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

`public interface Runnable`

The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called `run`.



Packages a piece of application functionality—as well as its parameterization in an object—to make it usable in another context

See www.dre.vanderbilt.edu/~schmidt/CommandProcessor.pdf

7.6 The Command Pattern

Week 7 · Software Design Patterns II

5 items

1 2 3 4

Command — GoF Object Behavioral

Known uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- GNU Emacs
- Microsoft Office tools
- Java **`Runnable`** interface
  - **`Runnable`** can also be used to implement the *Command Processor* pattern

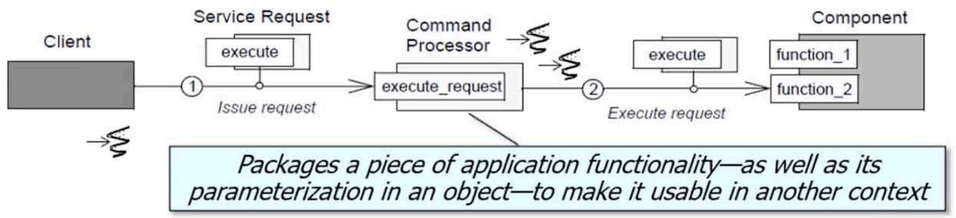
java.lang
Interface Runnable

All Known Subinterfaces:
RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:
AsyncBoxView.ChildState, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

```
public interface Runnable
```

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. The class must define a method of no arguments called run.



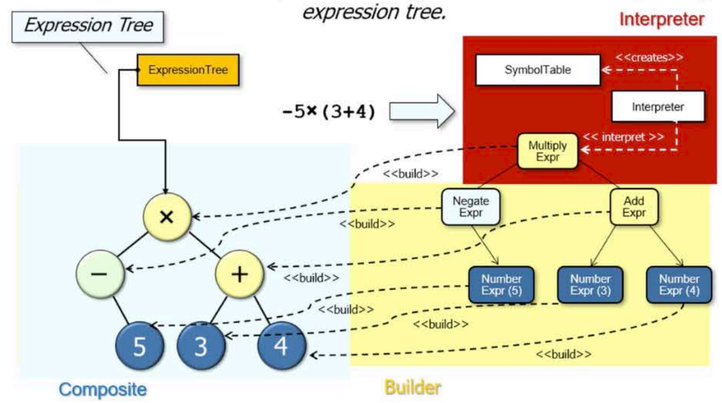
See www.dre.vanderbilt.edu/~schmidt/CommandProcessor.pdf

▸ **Transcript**

Week 7

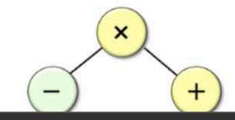
A Pattern for Flexibly Processing User Input

Purpose: *Process a user input expression and build a parse tree, which is then combined with other patterns and applied to build the corresponding expression tree.*



Context: OO Expression Tree Processing App

- The expression tree processing app also receives input in a variety of formats.
  - E.g., pre-order, in-order, level-order, post-order, etc.



Elements | Console | Sources | Network | 1

top | Filter | Default levels | 4 hidden

60 messages
afra... 4
ma... 56
59 user me...
No errors
1 warning
59 info
No verbose

Hide network | Log XMLHttpRequests
Preserve log | Eager evaluation
Selected context only | Autocomplete from history
Group similar | Evaluate triggers user activation

56 App.js:40