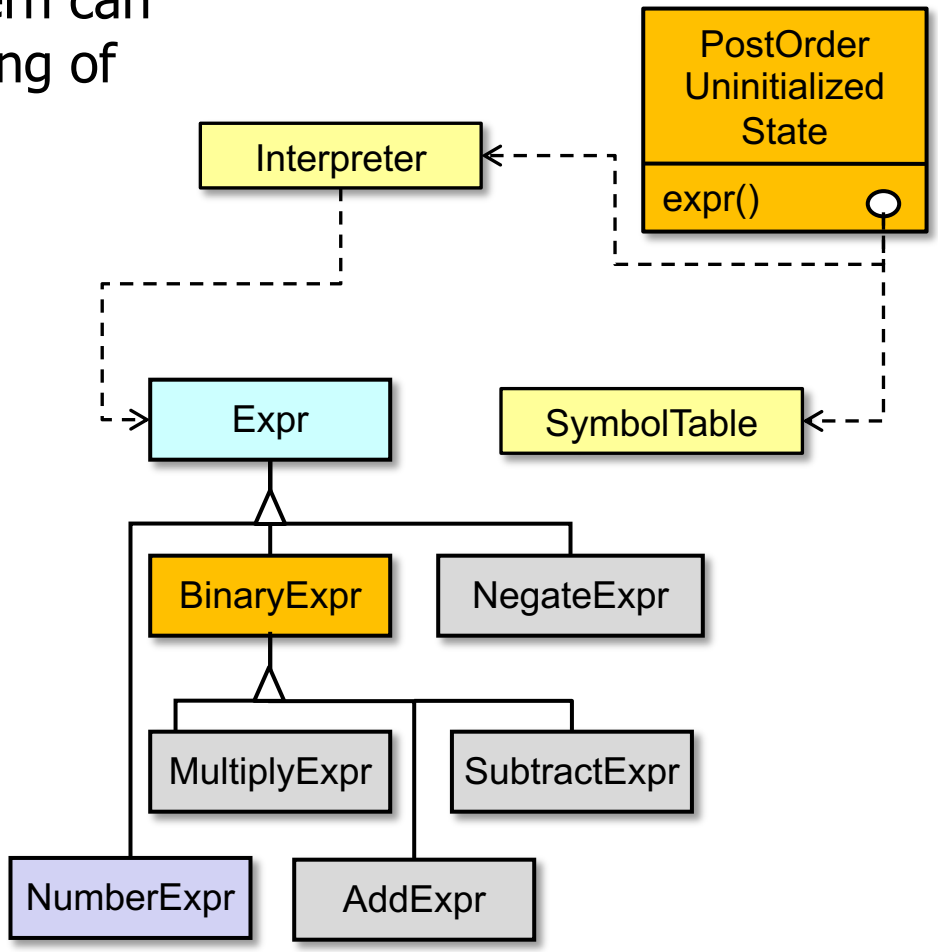
The Interpreter Pattern

Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Interpreter* pattern can be applied to automate the processing of input expressions from users in the expression tree processing app.

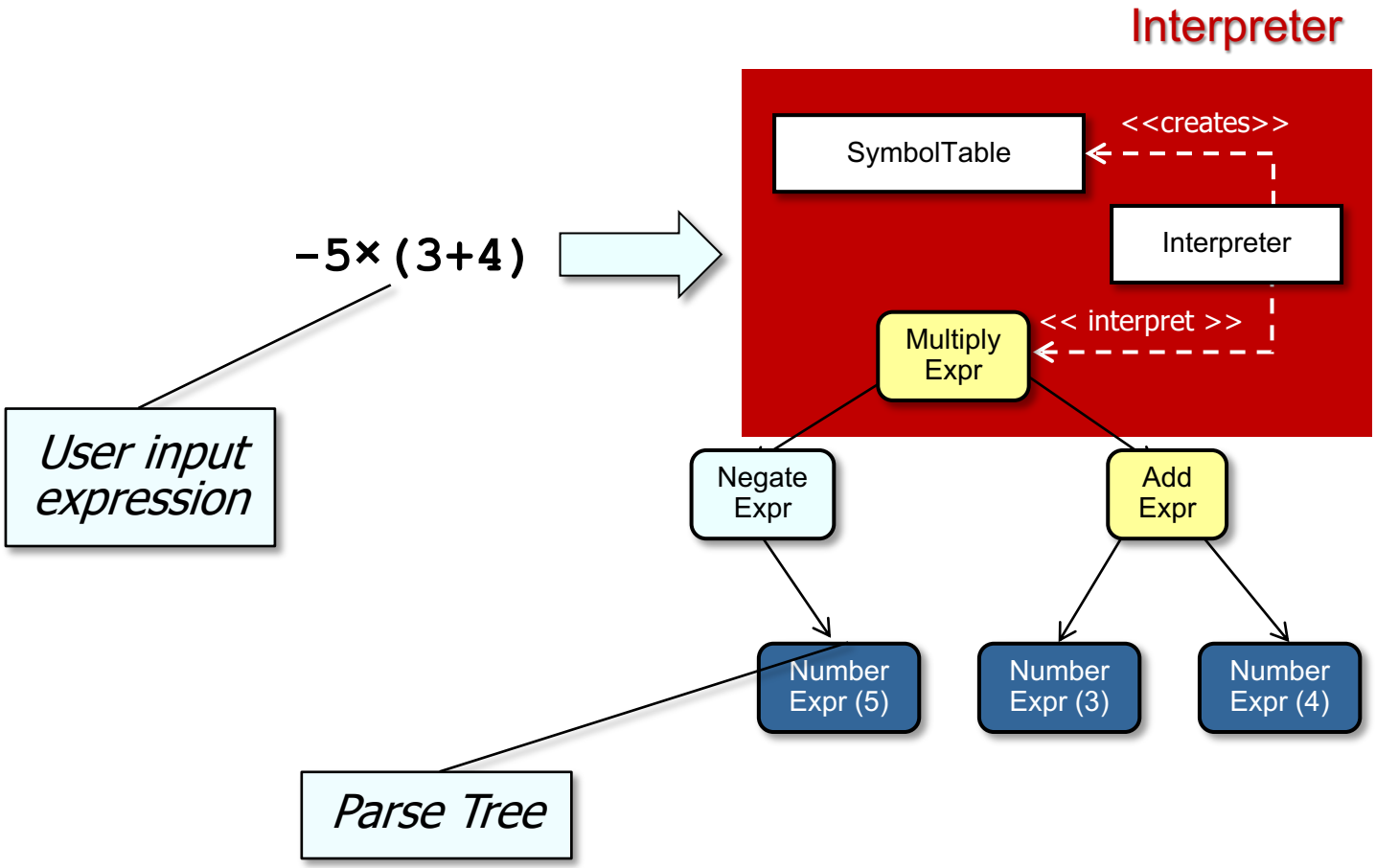


Douglas C. Schmidt

Motivating the Need for the Interpreter Pattern in the Expression Tree App

A Pattern for Flexibly Processing User Input

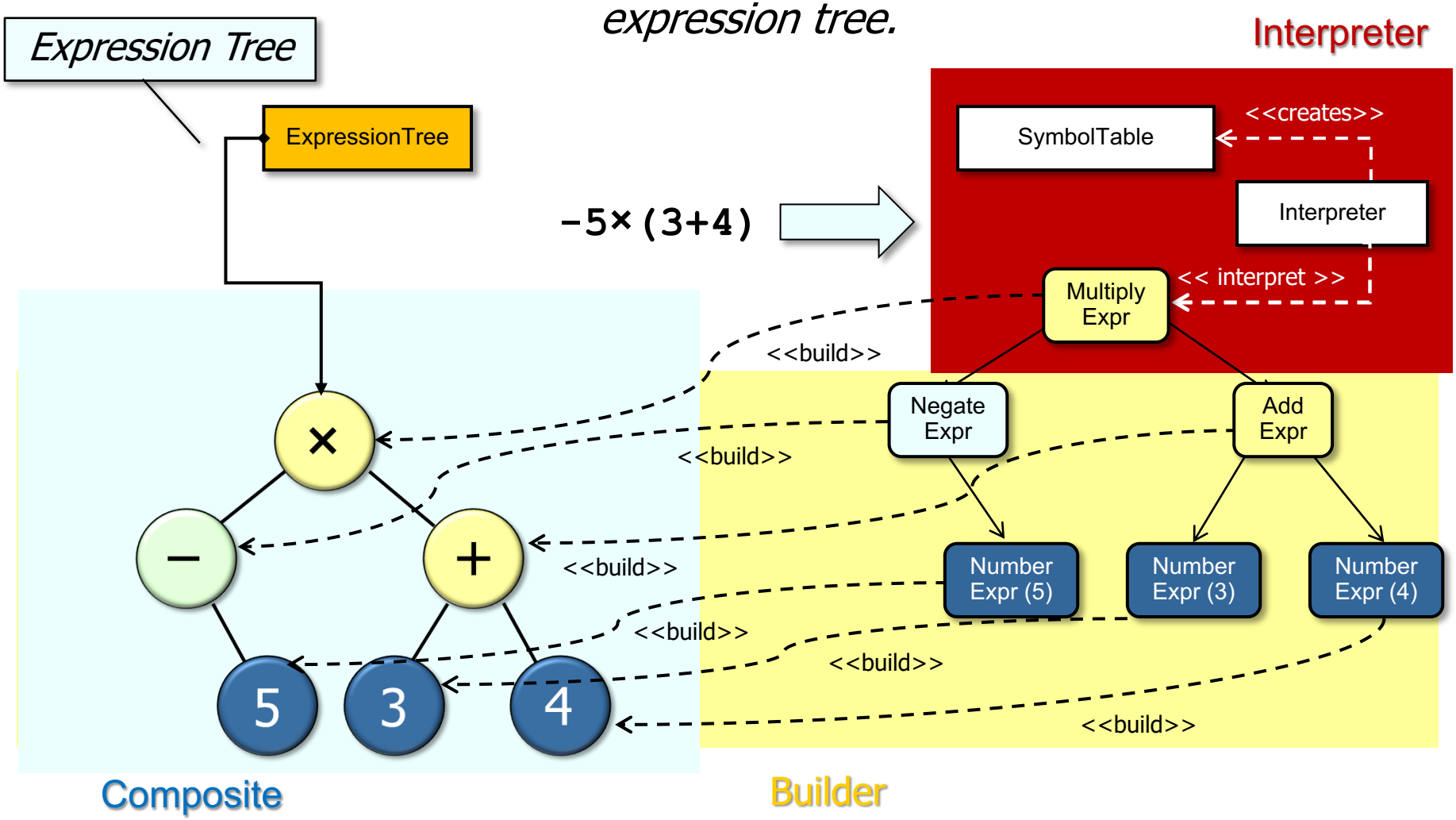
***Purpose**: Process a user input expression and build a parse tree*



Interpreter automates the parsing of input expressions from users.

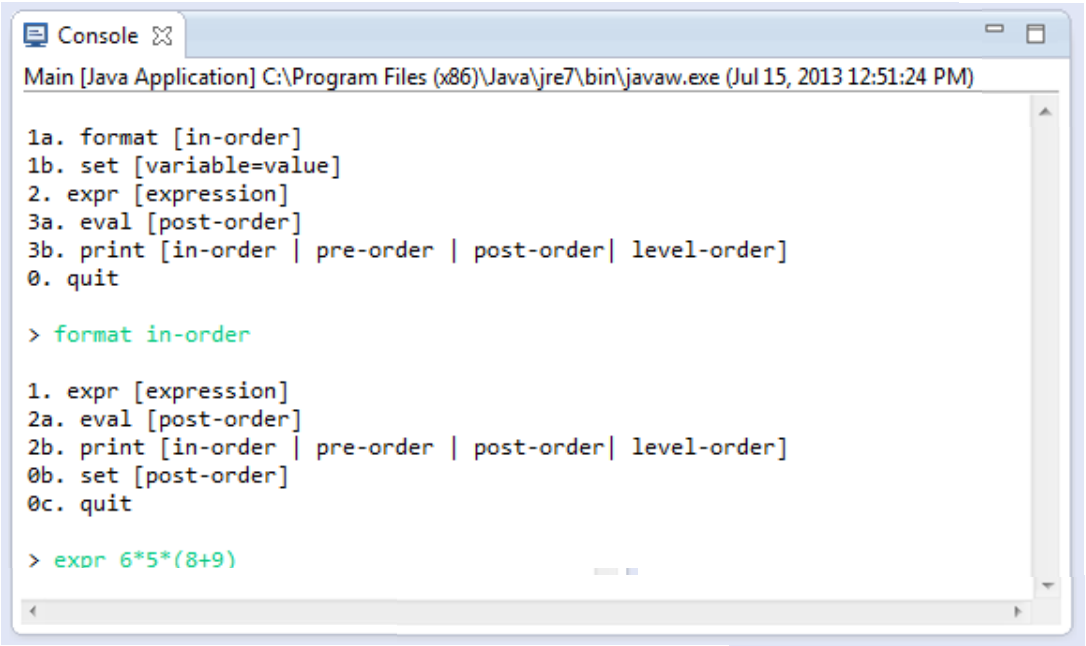
A Pattern for Flexibly Processing User Input

***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.*



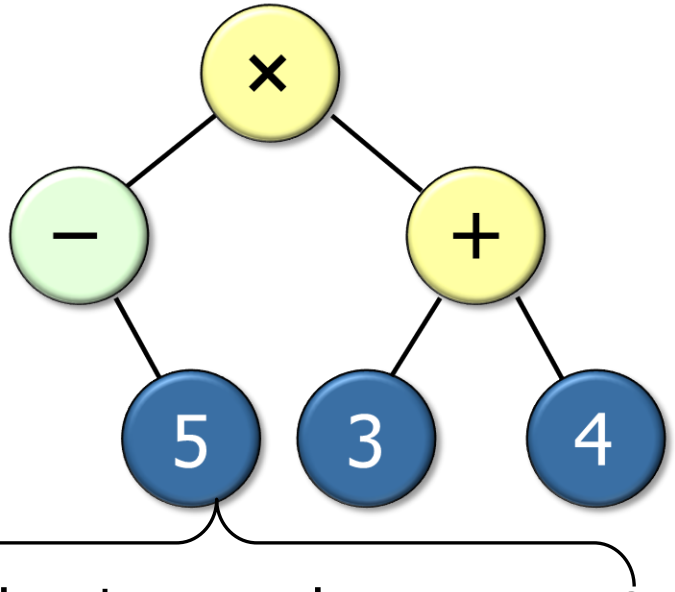
Context: OO Expression Tree Processing App

- The expression tree processing app receives input from a variety of sources.
  - e.g., command-line, GUI, etc.



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`

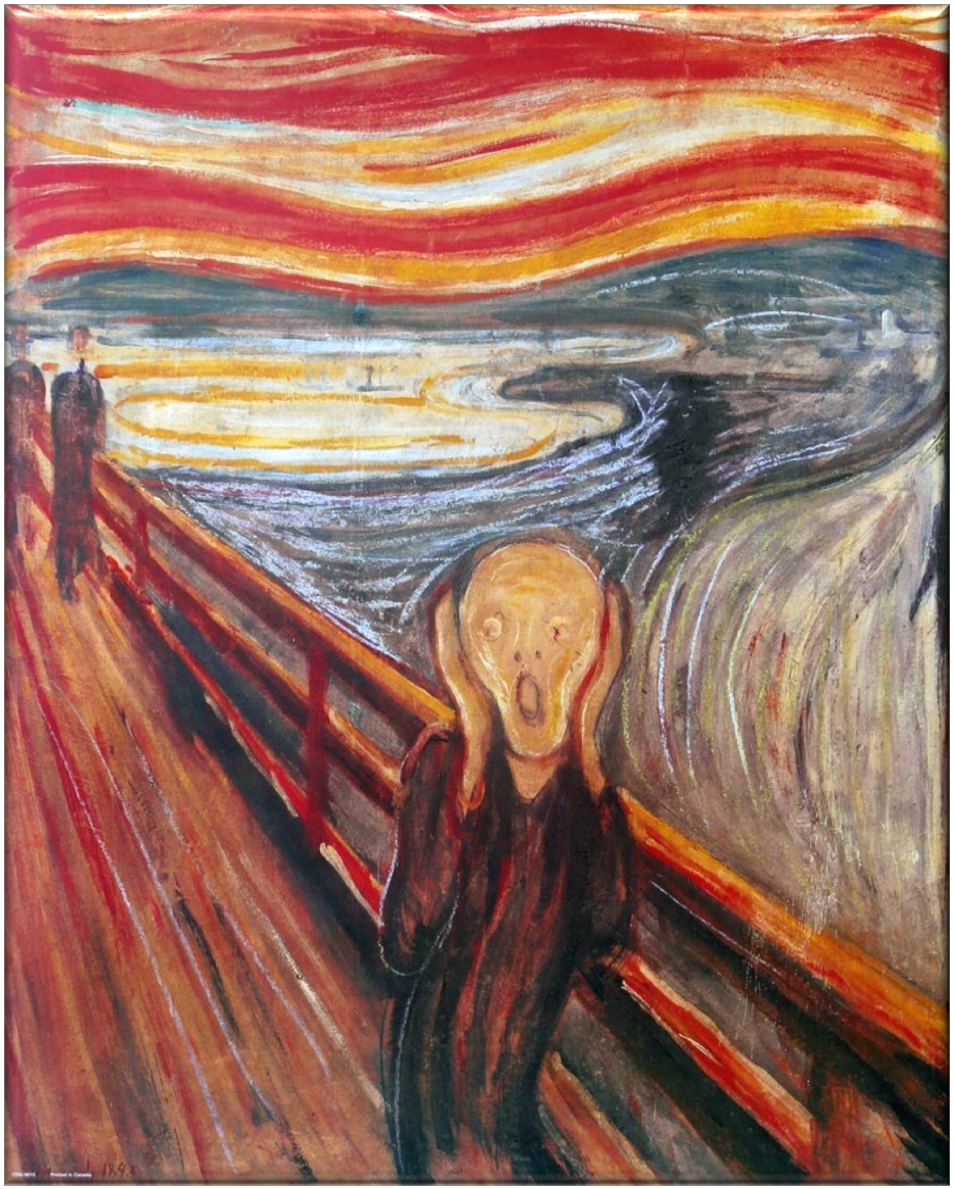
"Post-order" input expression = `5~34+×`

"Level-order" input expression = `×-+534`

"In-order" input expression = `-5×(3+4)`

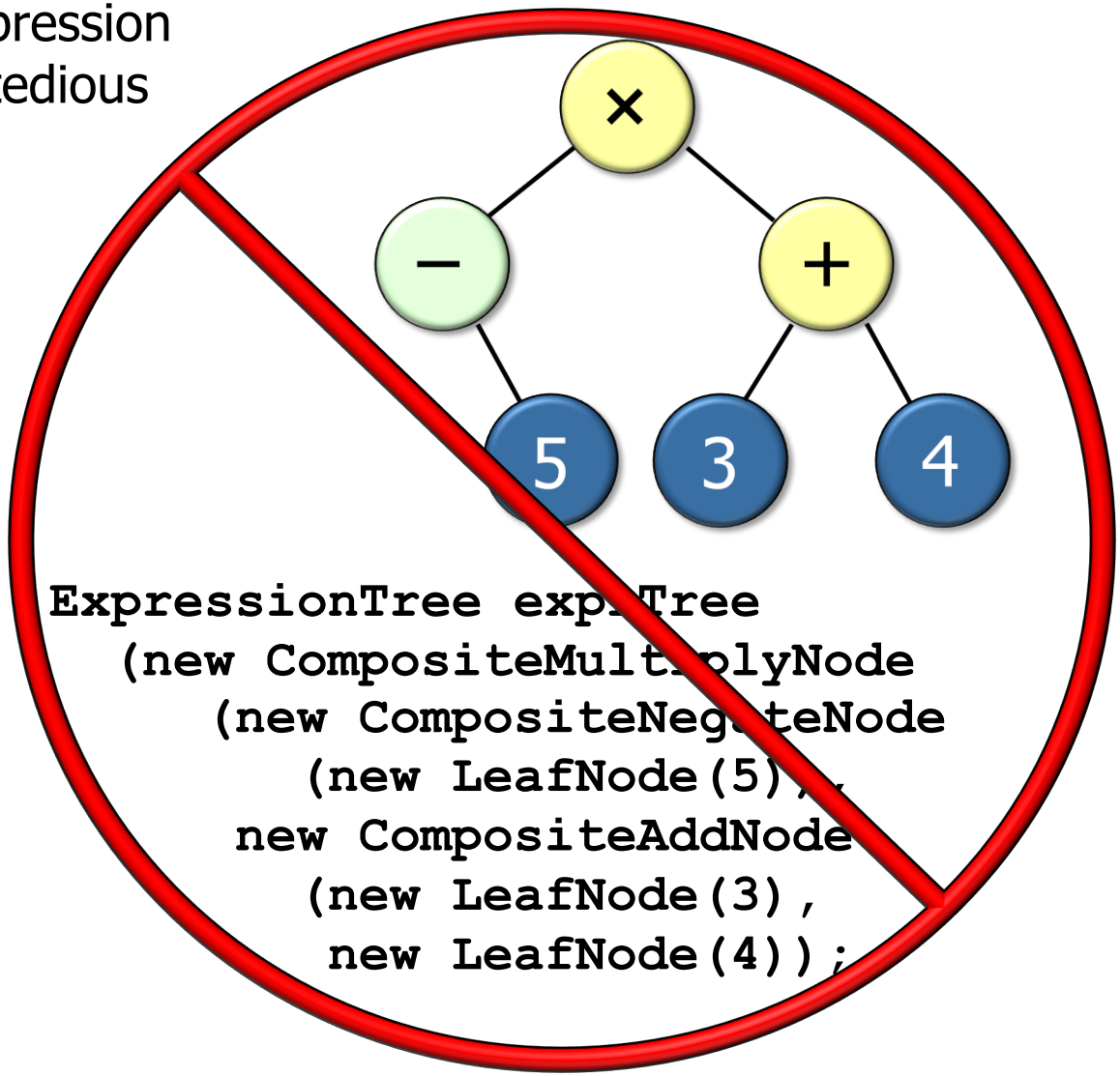
Problem: Inflexible Expression Input Processing

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



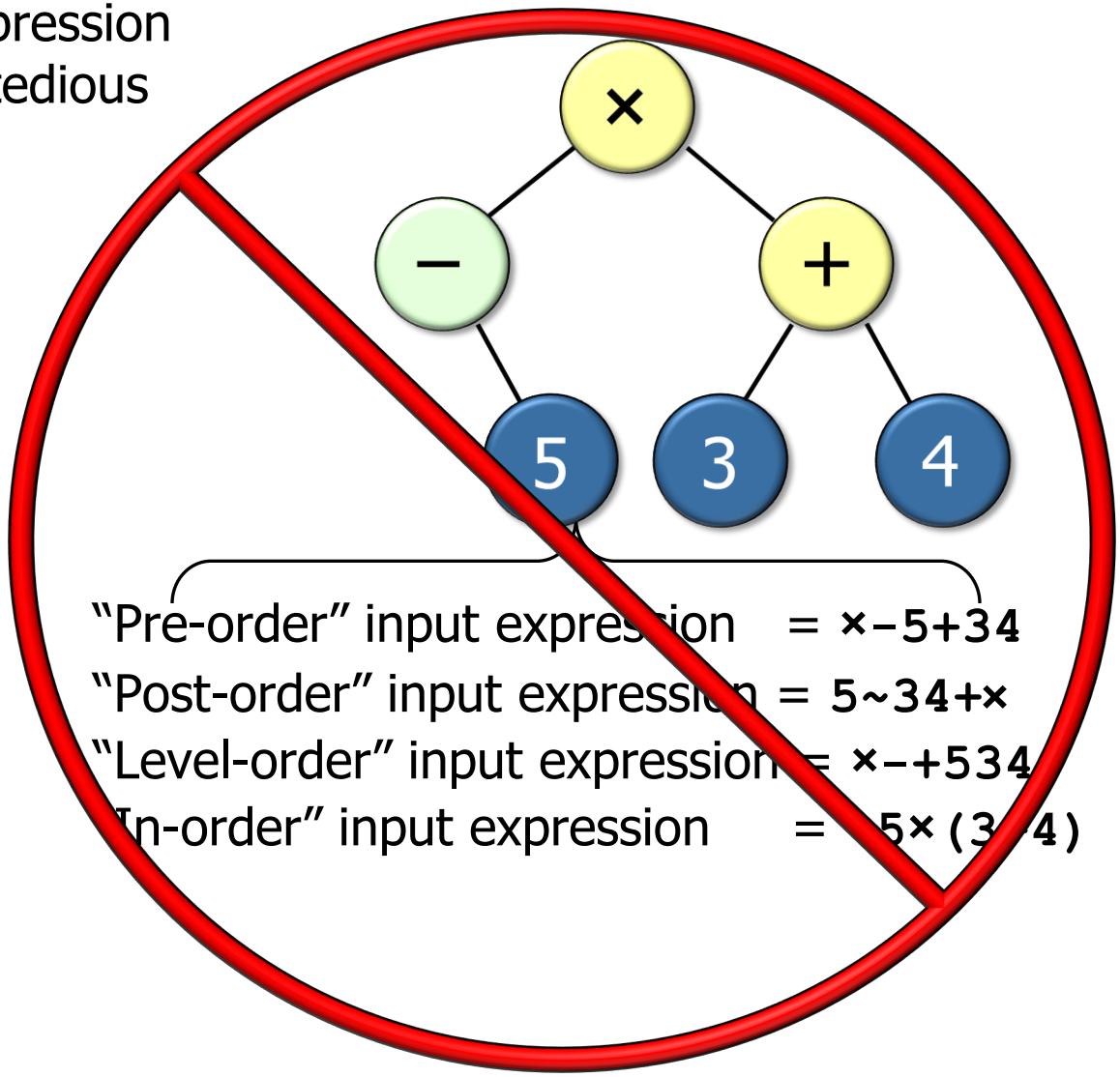
Problem: Inflexible Expression Input Processing

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



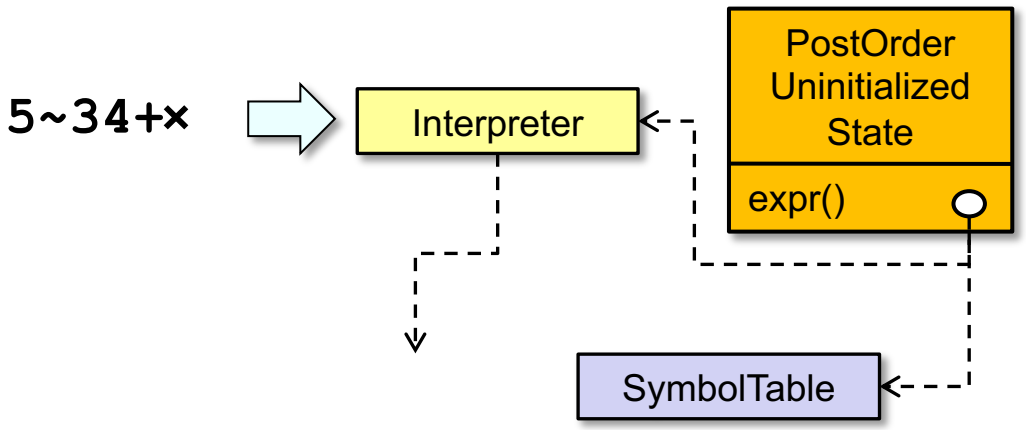
Problem: Inflexible Expression Input Processing

- 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.



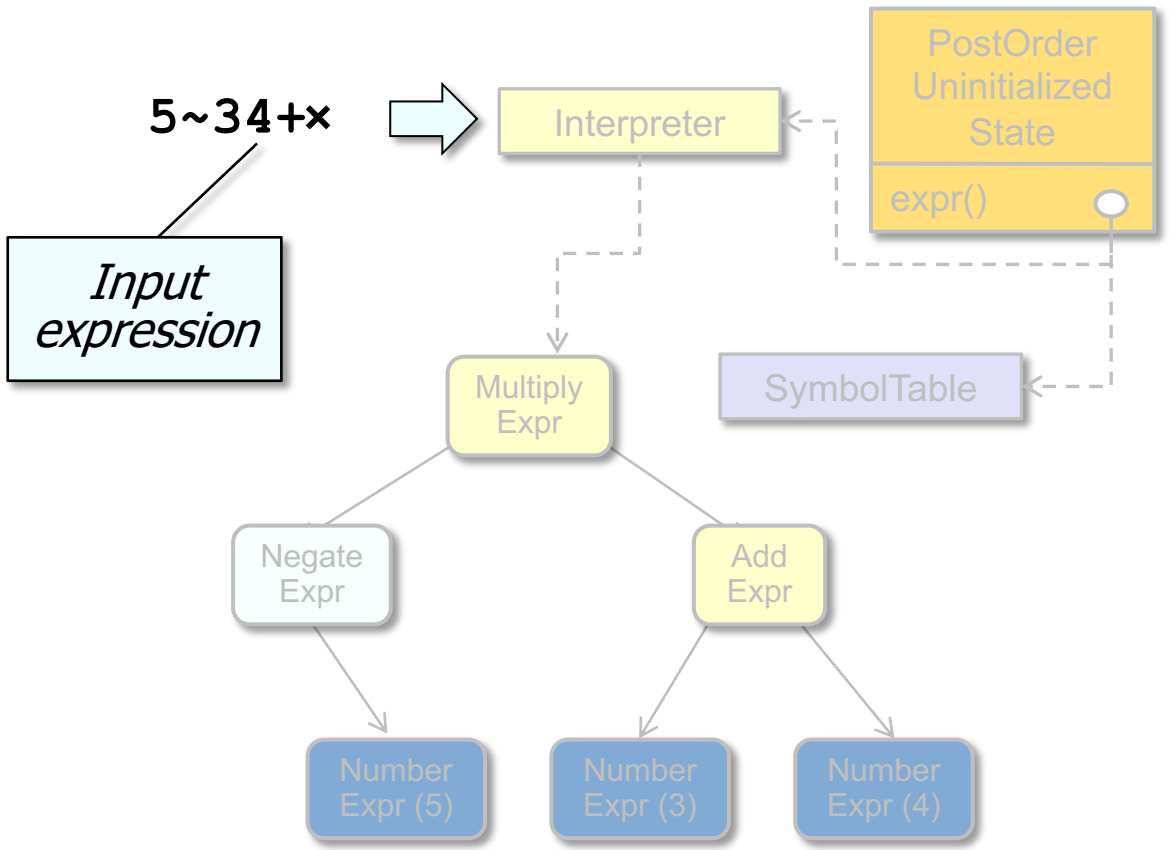
Solution: Create Interpreter to Process Input

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



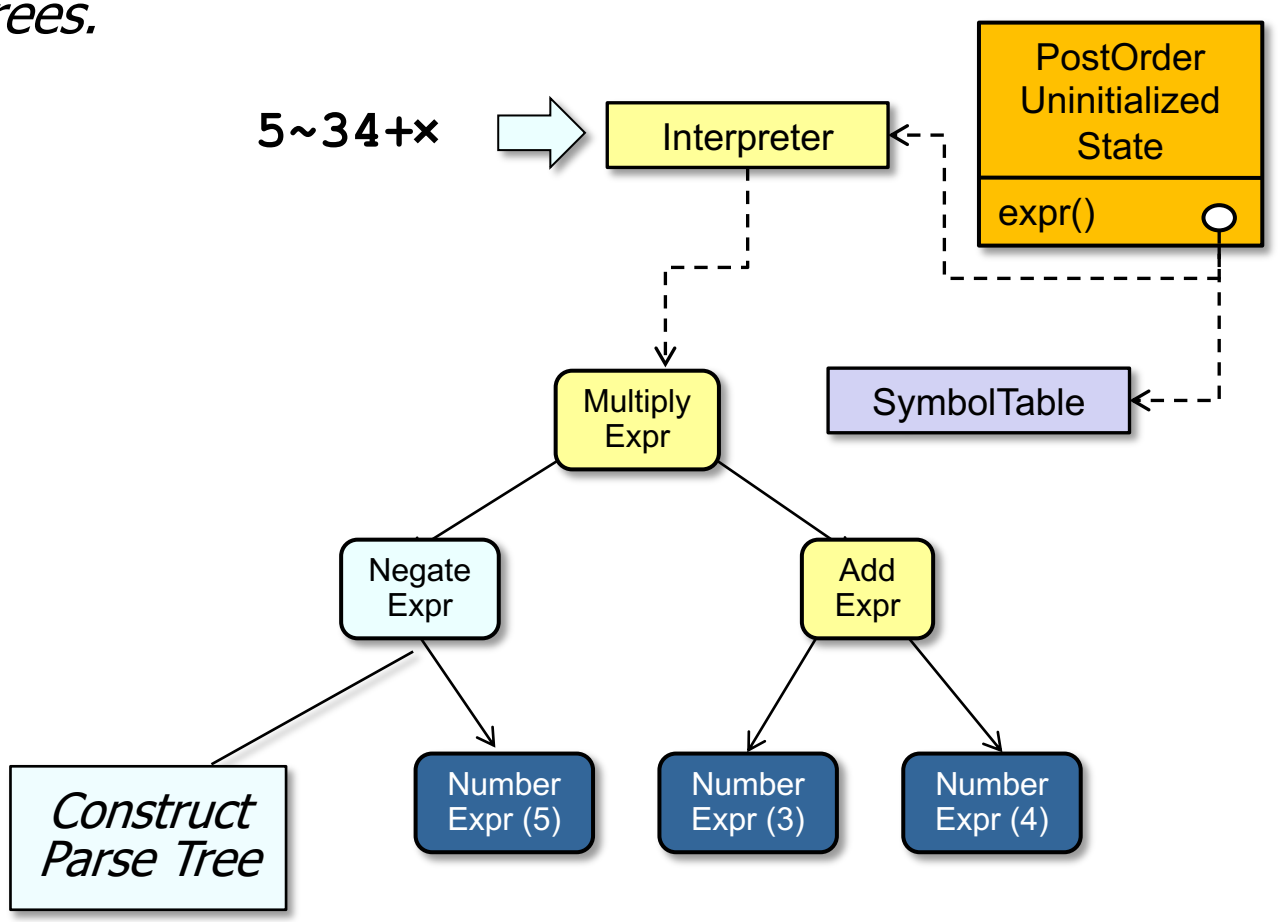
Solution: Create Interpreter to Process Input

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



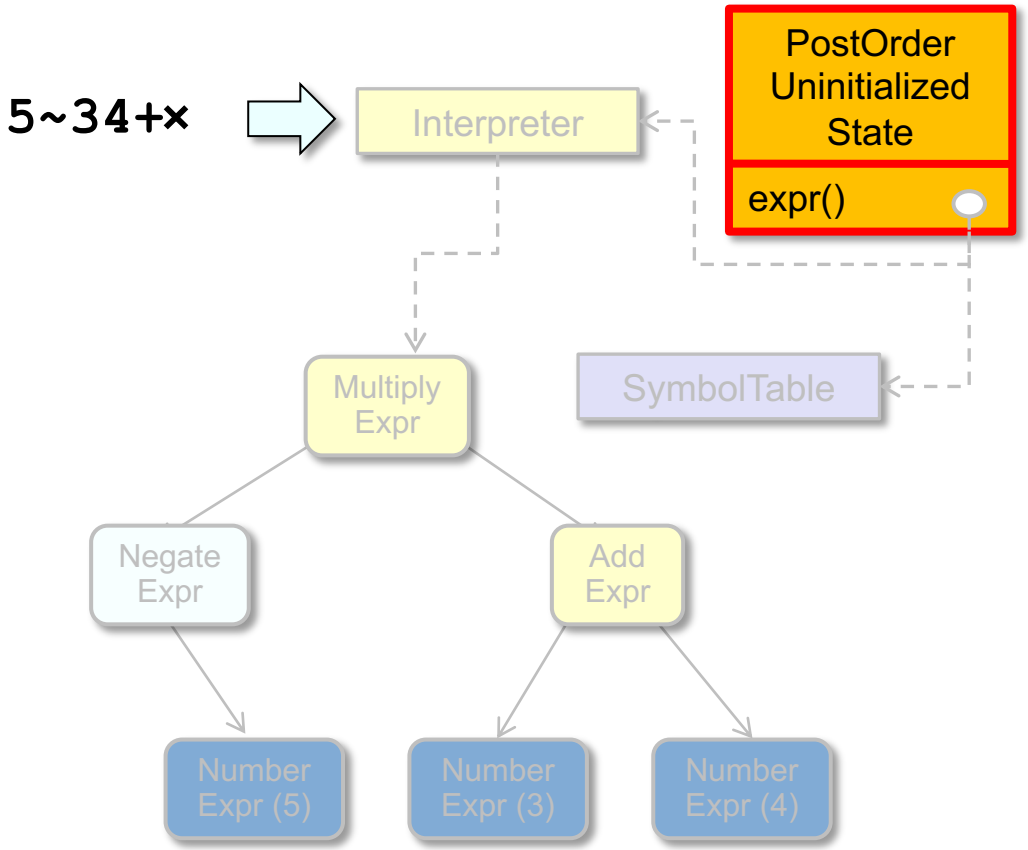
Solution: Create Interpreter to Process Input

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



Solution: Create Interpreter to Process Input

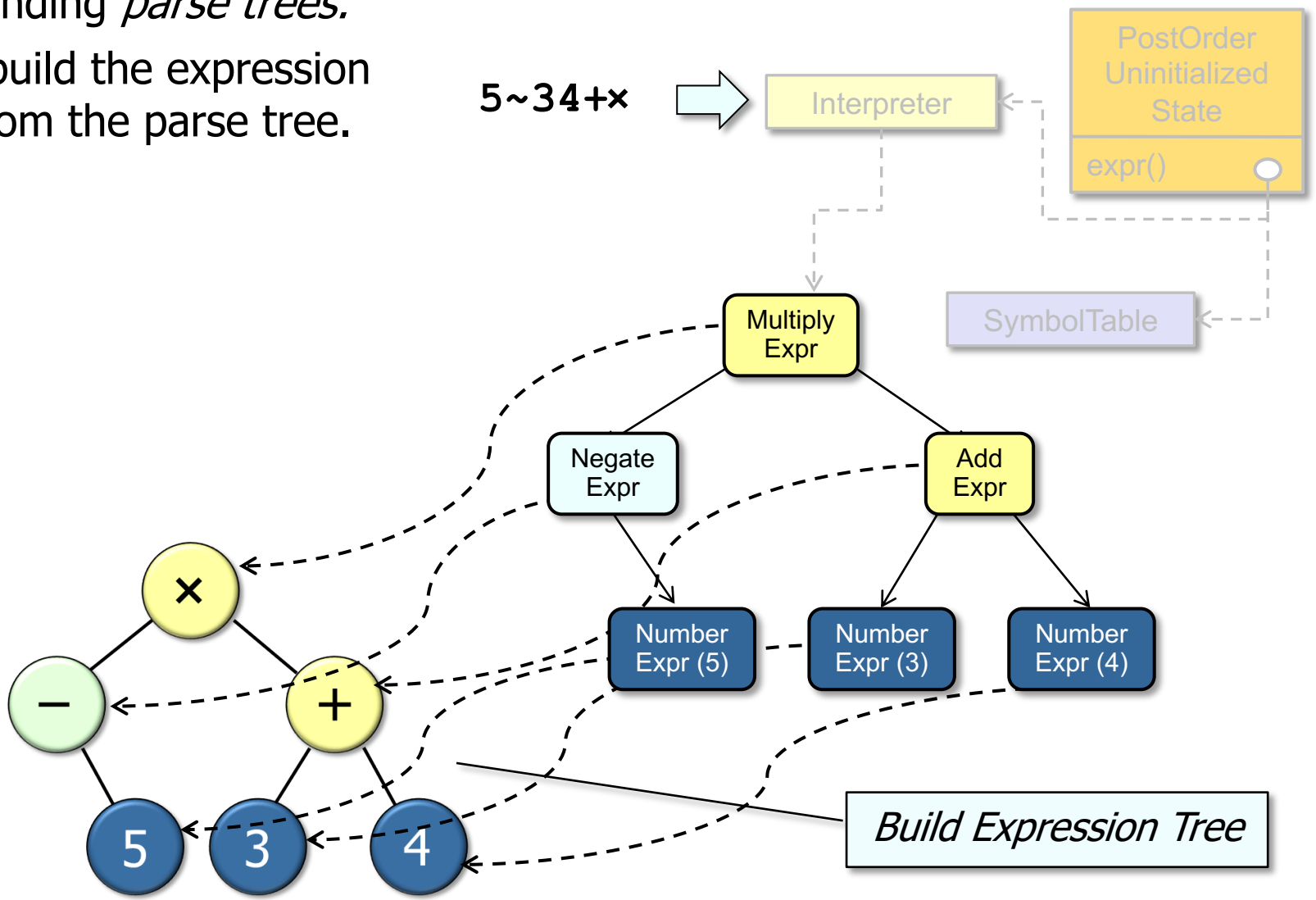
- Create an interpreter that processes user input expressions and creates the corresponding *parse trees.*



The **`PostOrderUninitializedState`** class is covered in the *State* pattern lesson.

Solution: Create Interpreter to Process Input

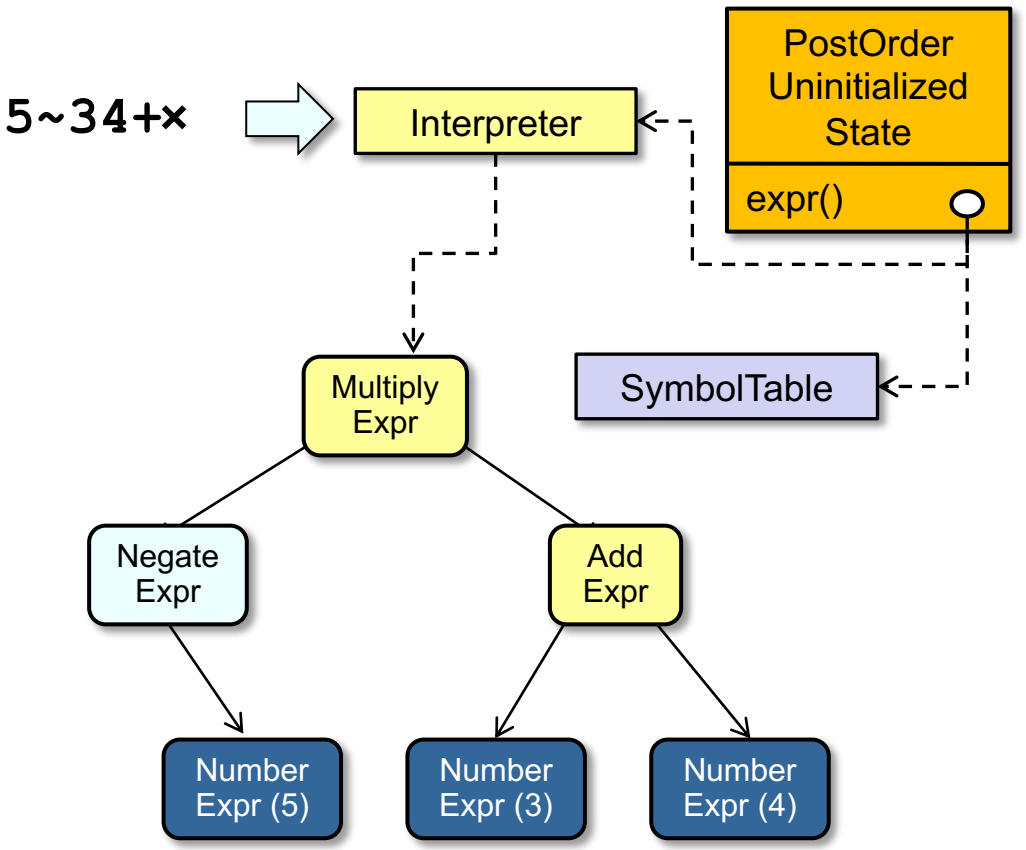
- Create an interpreter that processes user input expressions and creates the corresponding *parse trees.*
  - Then build the expression tree from the parse tree.



We'll describe how to apply the *Builder* pattern in the next lesson.

Solution: Create Interpreter to Process Input

- Create an interpreter that processes user input expressions and creates the corresponding *parse trees.*
  - Then build the expression tree from the parse tree.

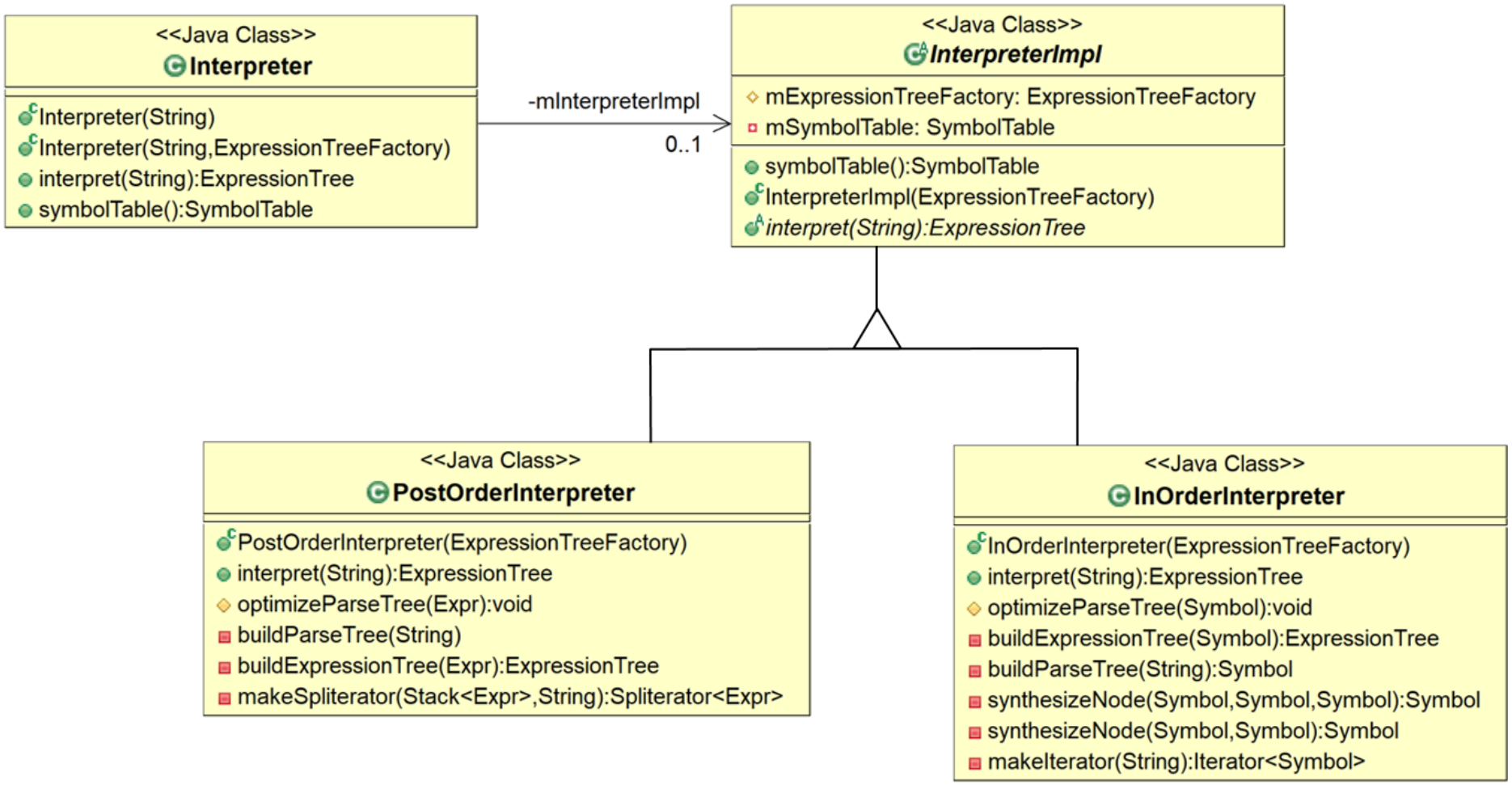


We just focus on using *Interpreter* to build the parse tree in this lesson.

Interpreter Class Overview

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

Interpreter class hierarchy

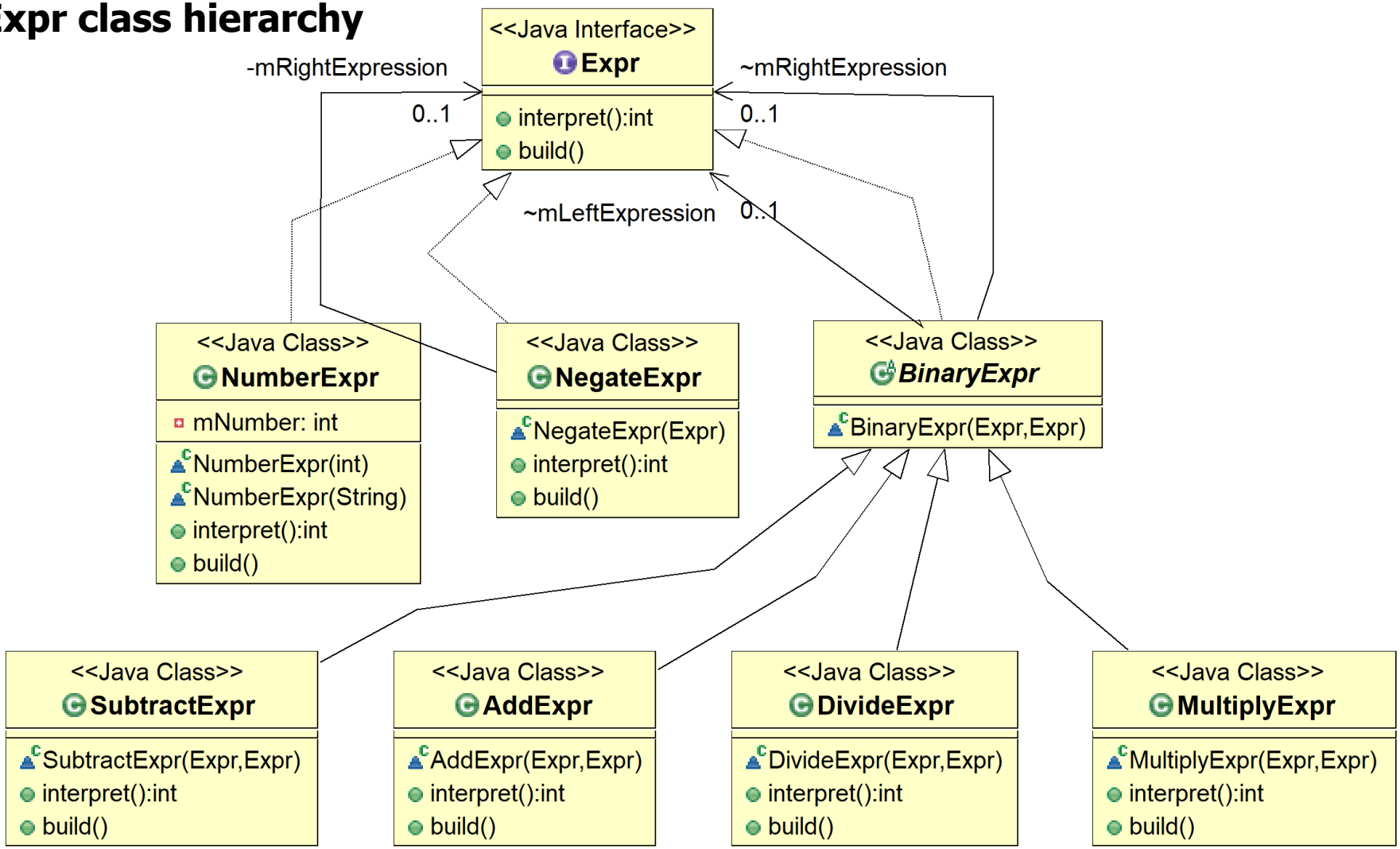


The *Bridge* pattern is used to support multiple interpreter implementations.

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)
```

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)
```

Controls the user input expression interpretation steps

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)
```

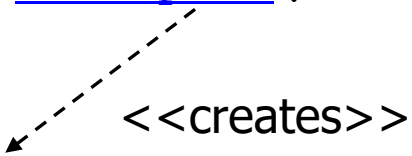
Hook methods for optimization and generation

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()
```

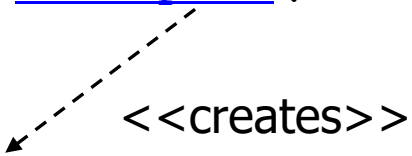
`SymbolTable` can be used to implement setters/getters for variable leaf nodes

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

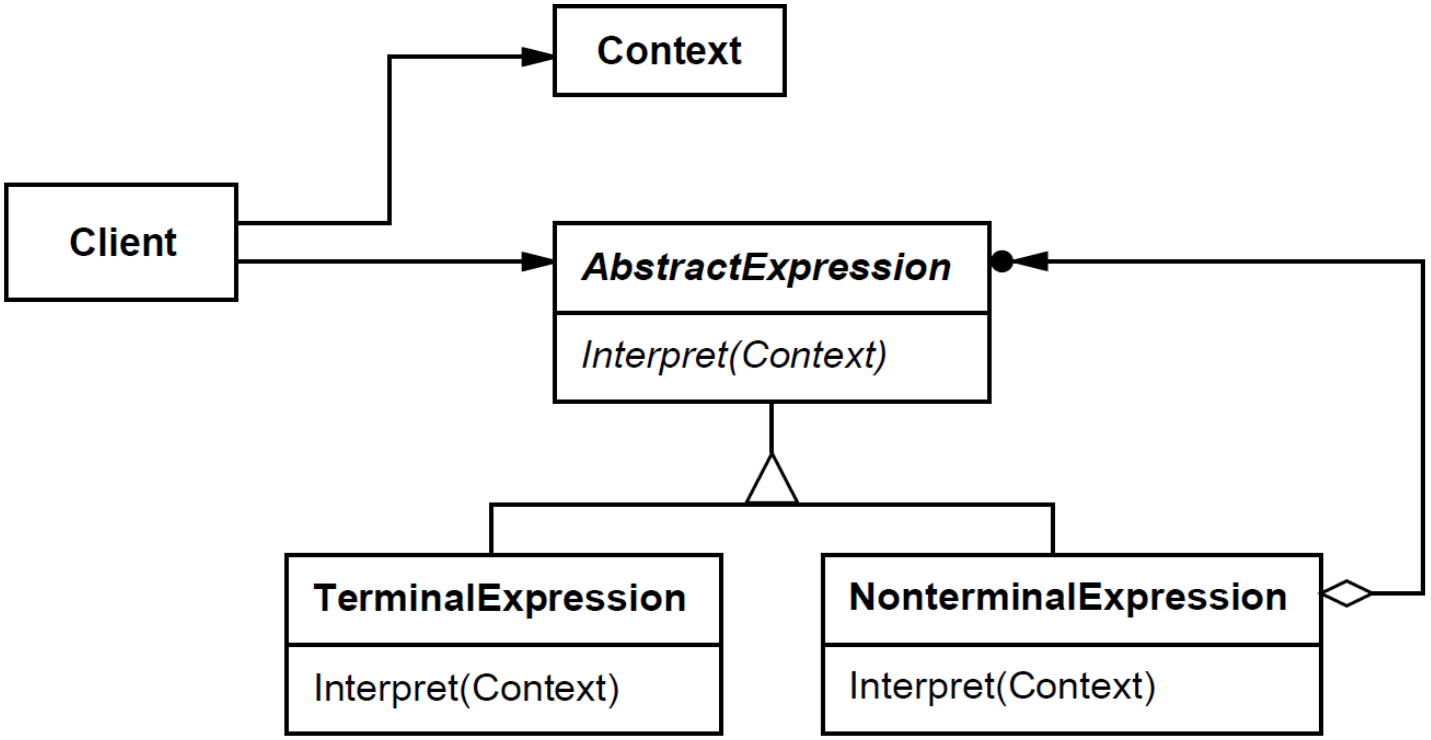
The Interpreter Pattern

Structure and Functionality

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Interpreter* pattern can be applied to automate the processing of input expressions from users.
- Understand the structure and functionality of the *Interpreter* pattern.



Douglas C. Schmidt

Structure and Functionality of the Interpreter Pattern

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

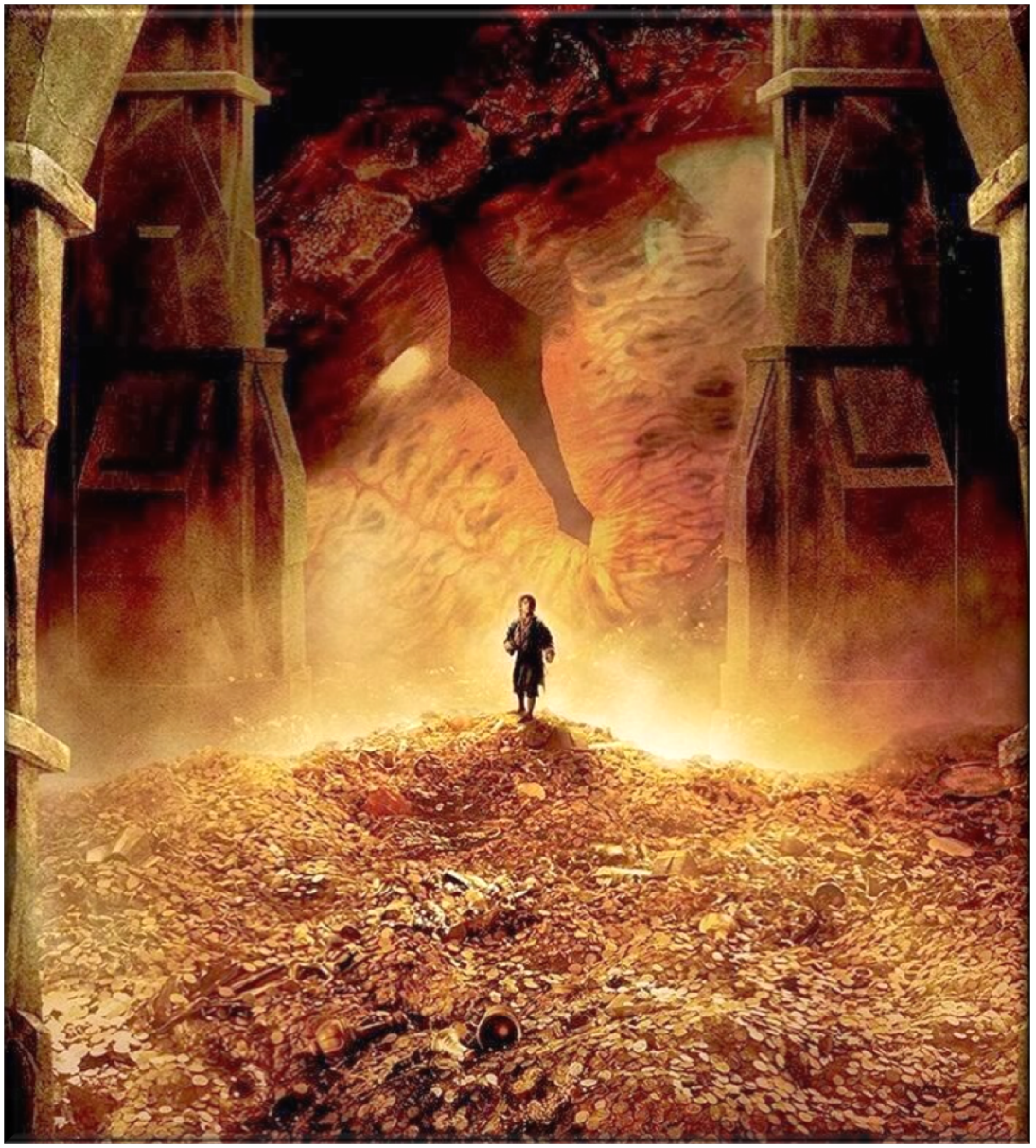
Applicability

- When the grammar is simple and relatively stable

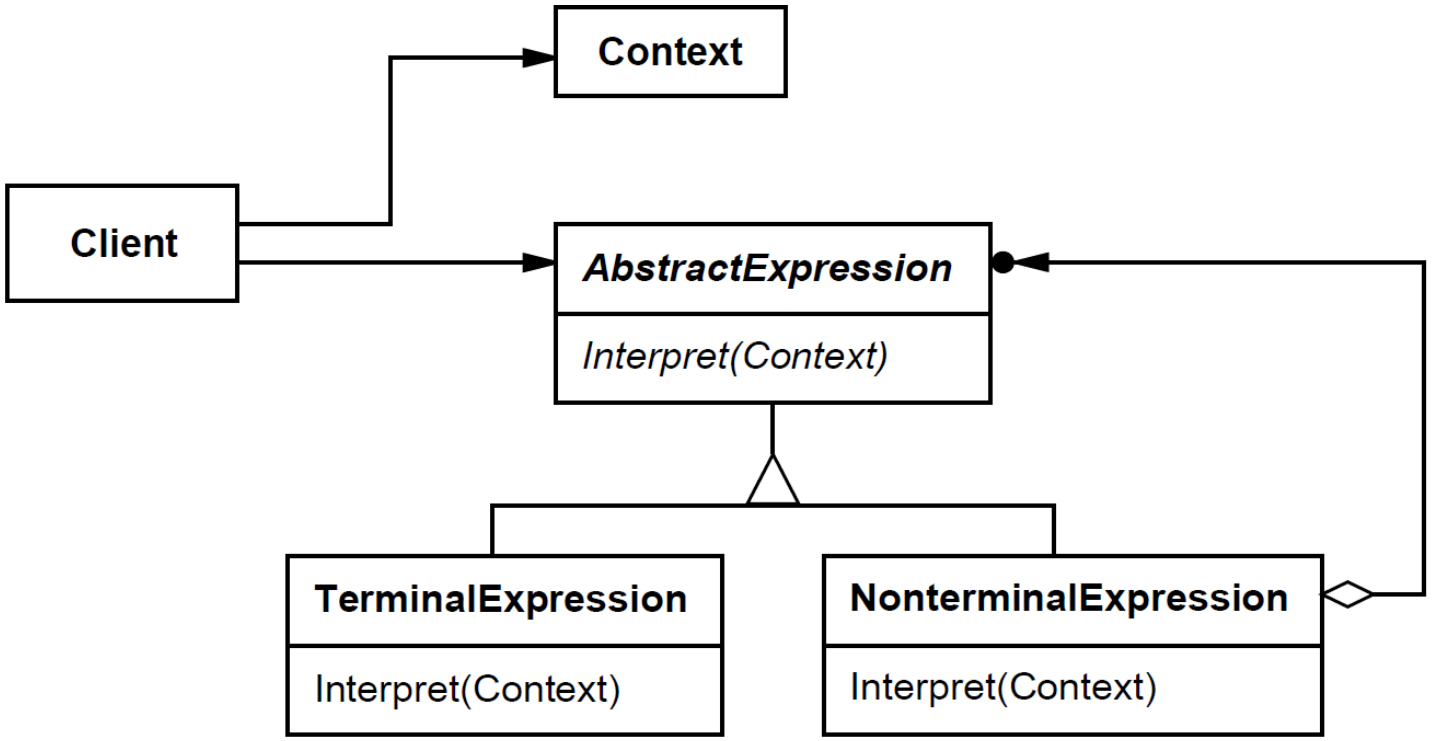
```
expr ::= factor expr-tail

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

factor ::= term factor-tail

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

mul_div ::= '×' | '/'

add_sub ::= '+' | '-`

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

Applicability

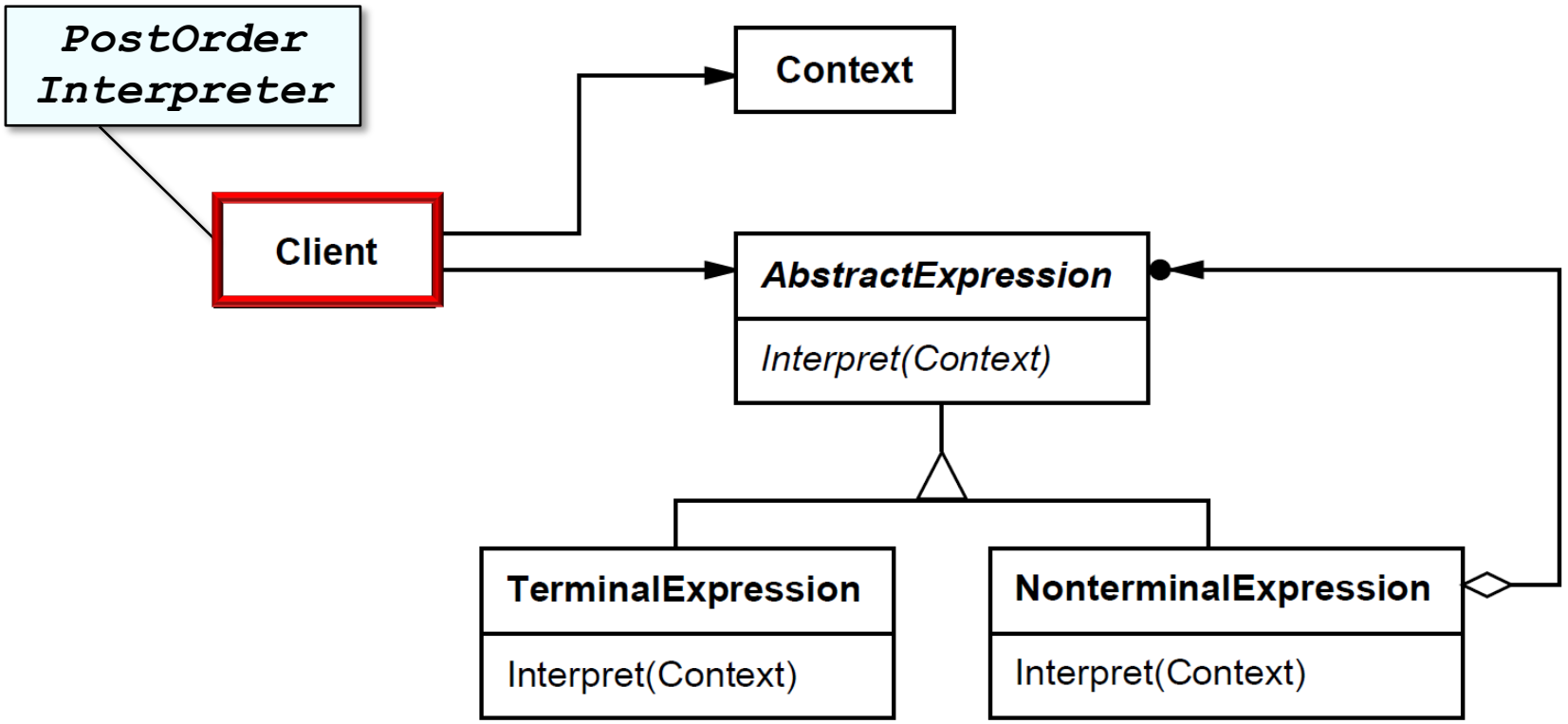
- When the grammar is simple and relatively stable
- When time/space efficiency is not a critical concern



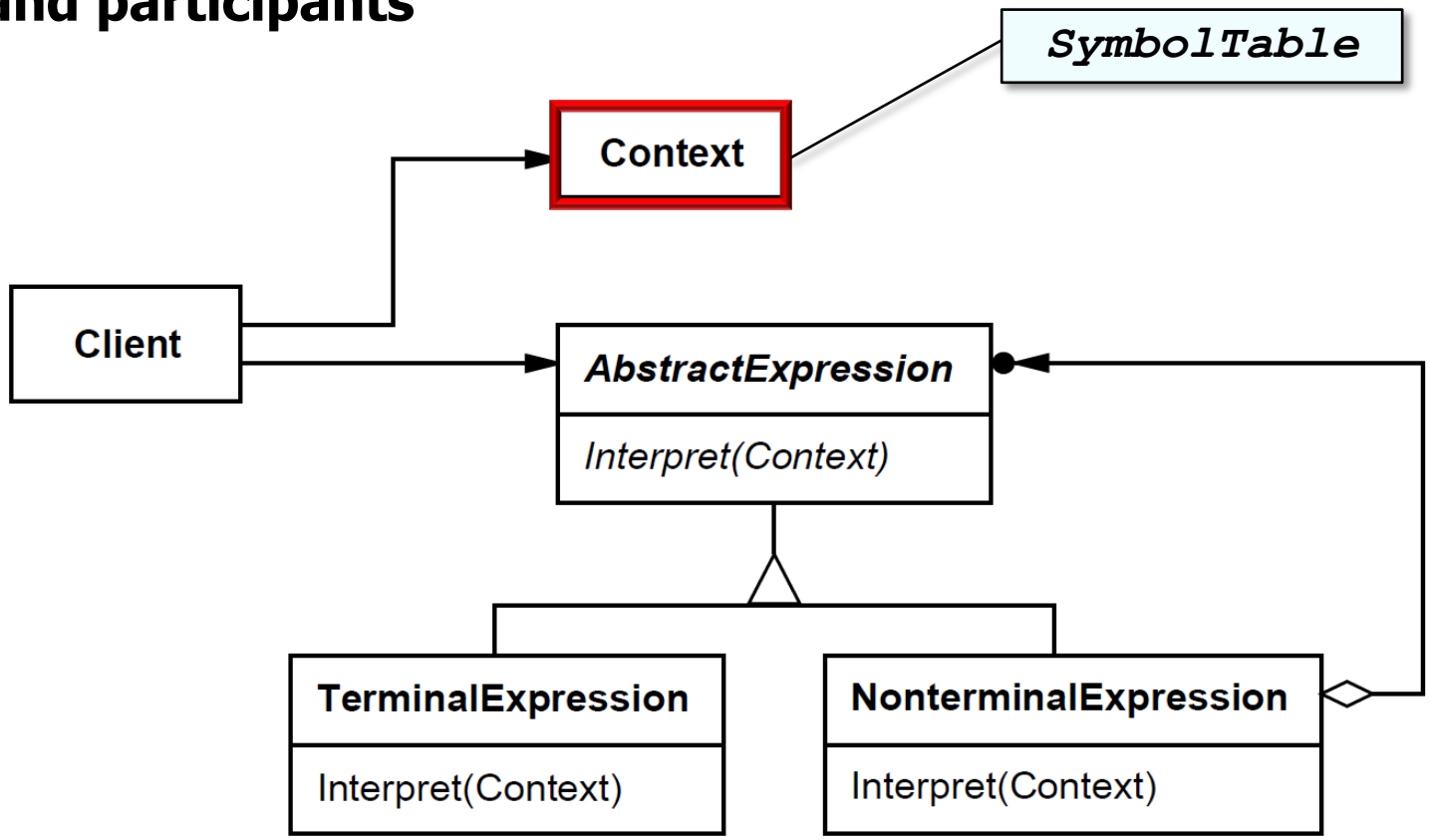
Structure and participants



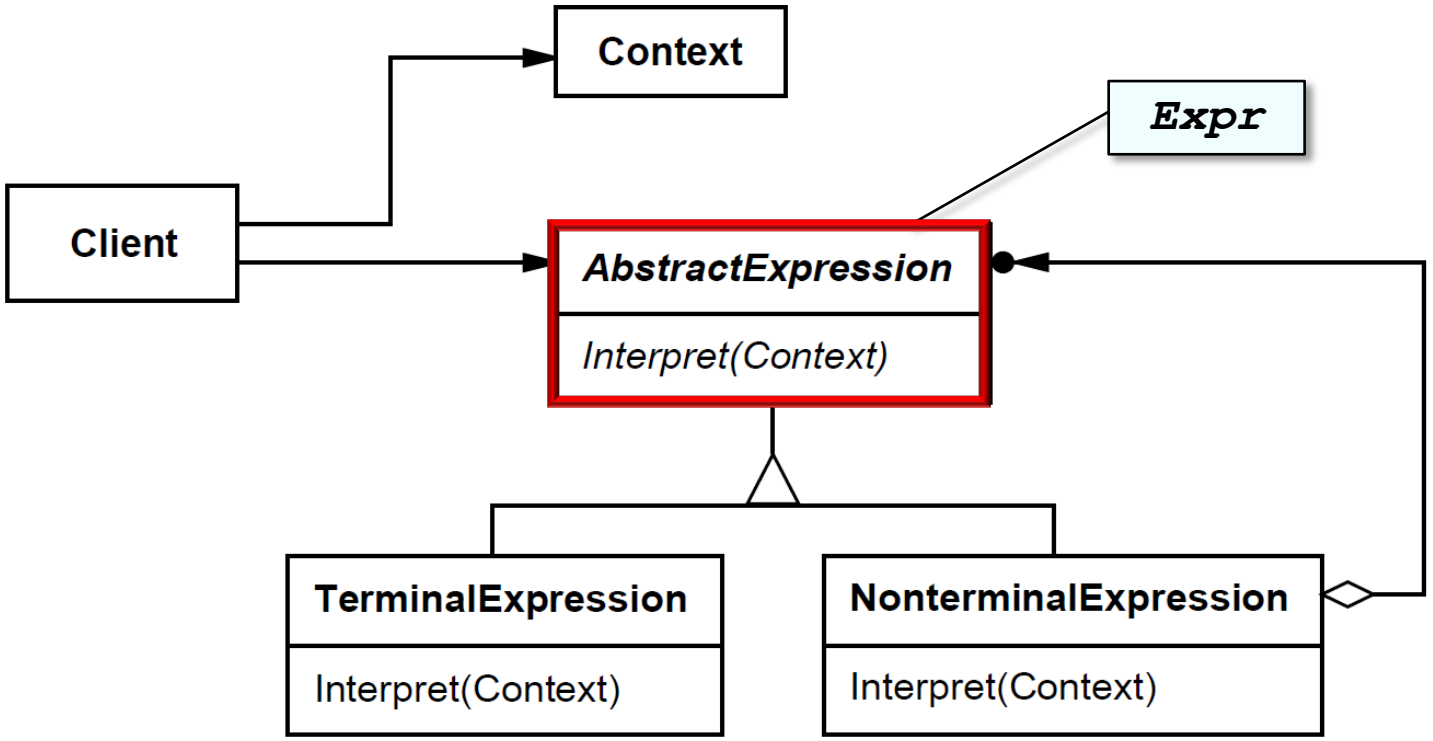
Structure and participants



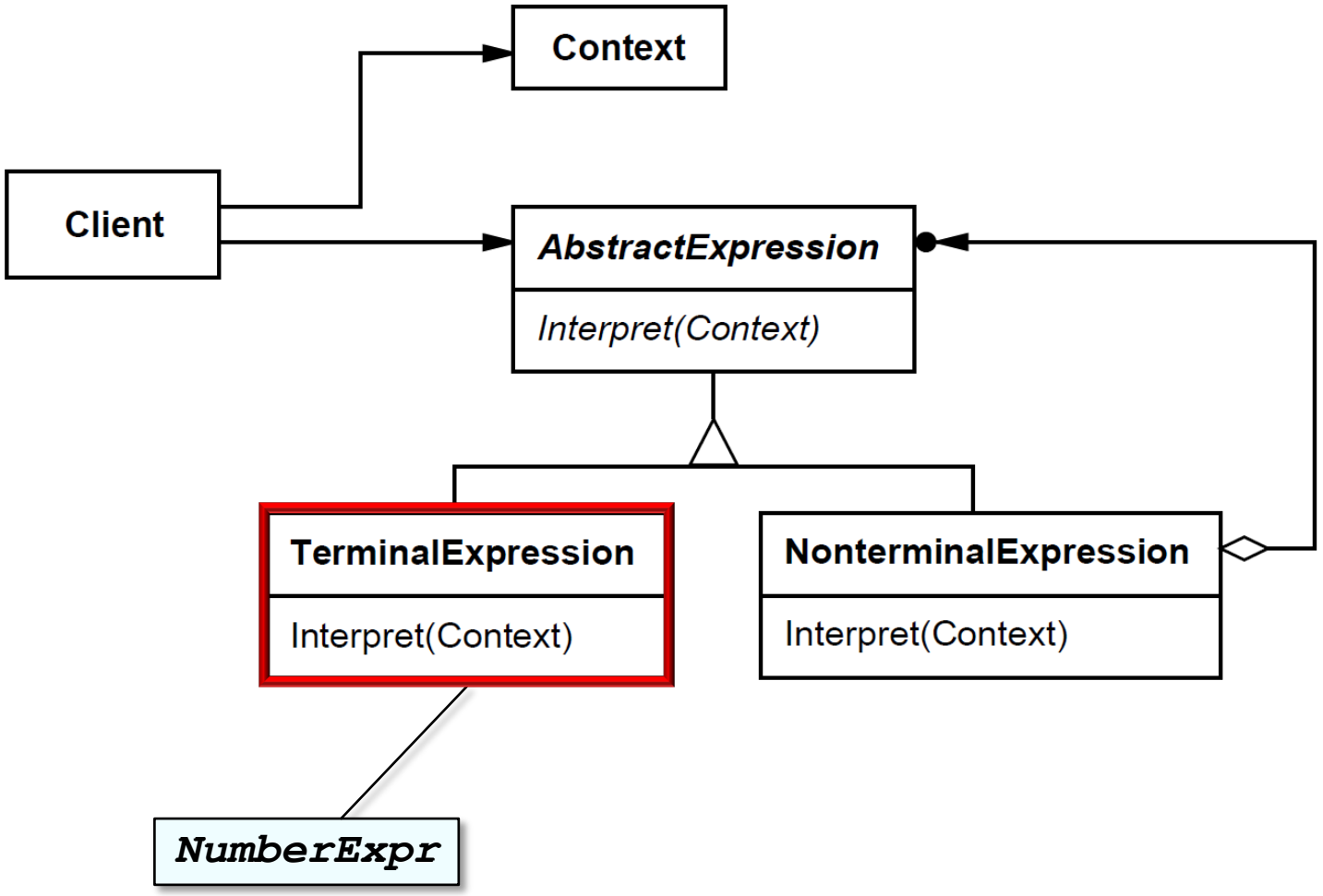
Structure and participants



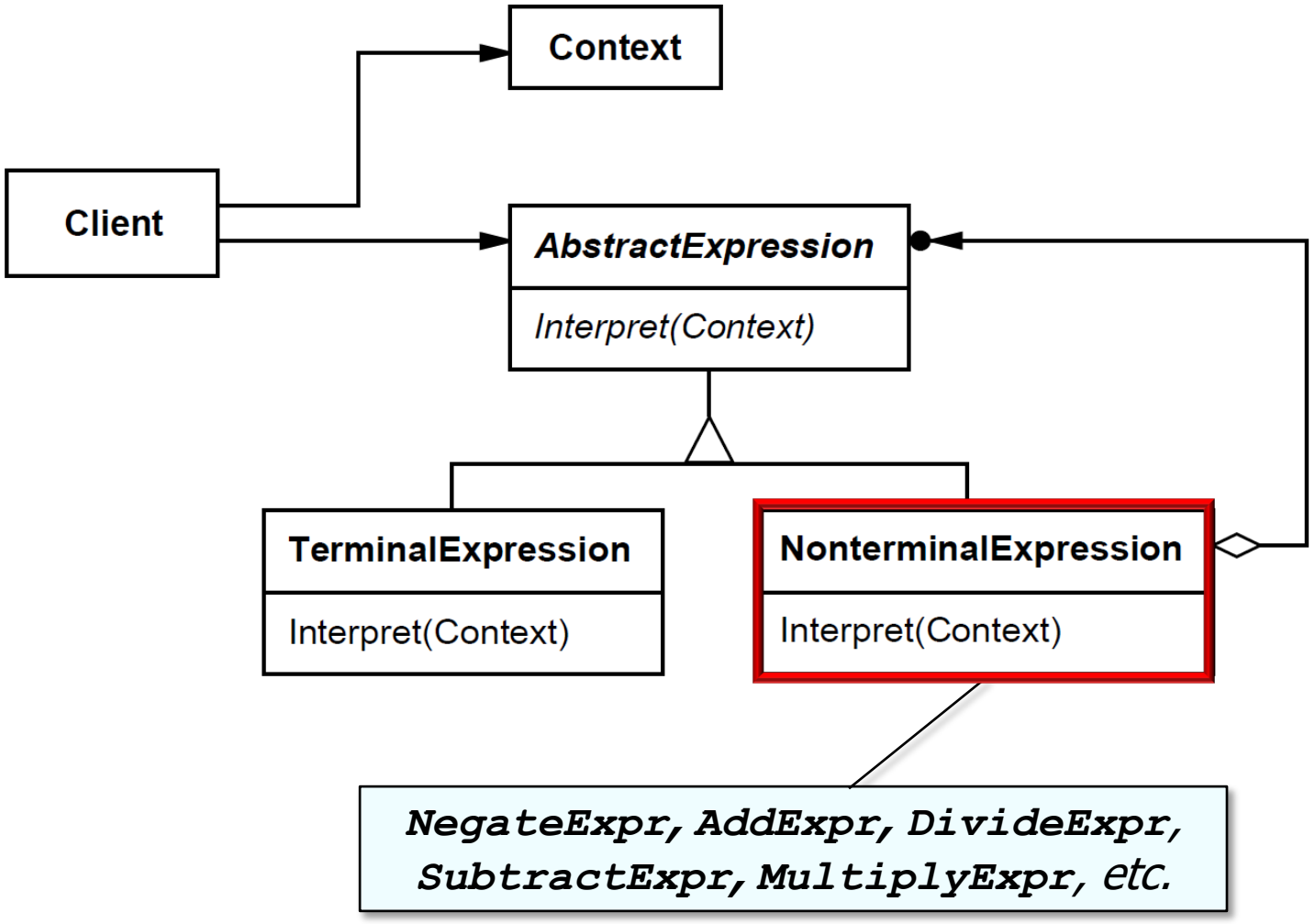
Structure and participants



Structure and participants



Structure and participants



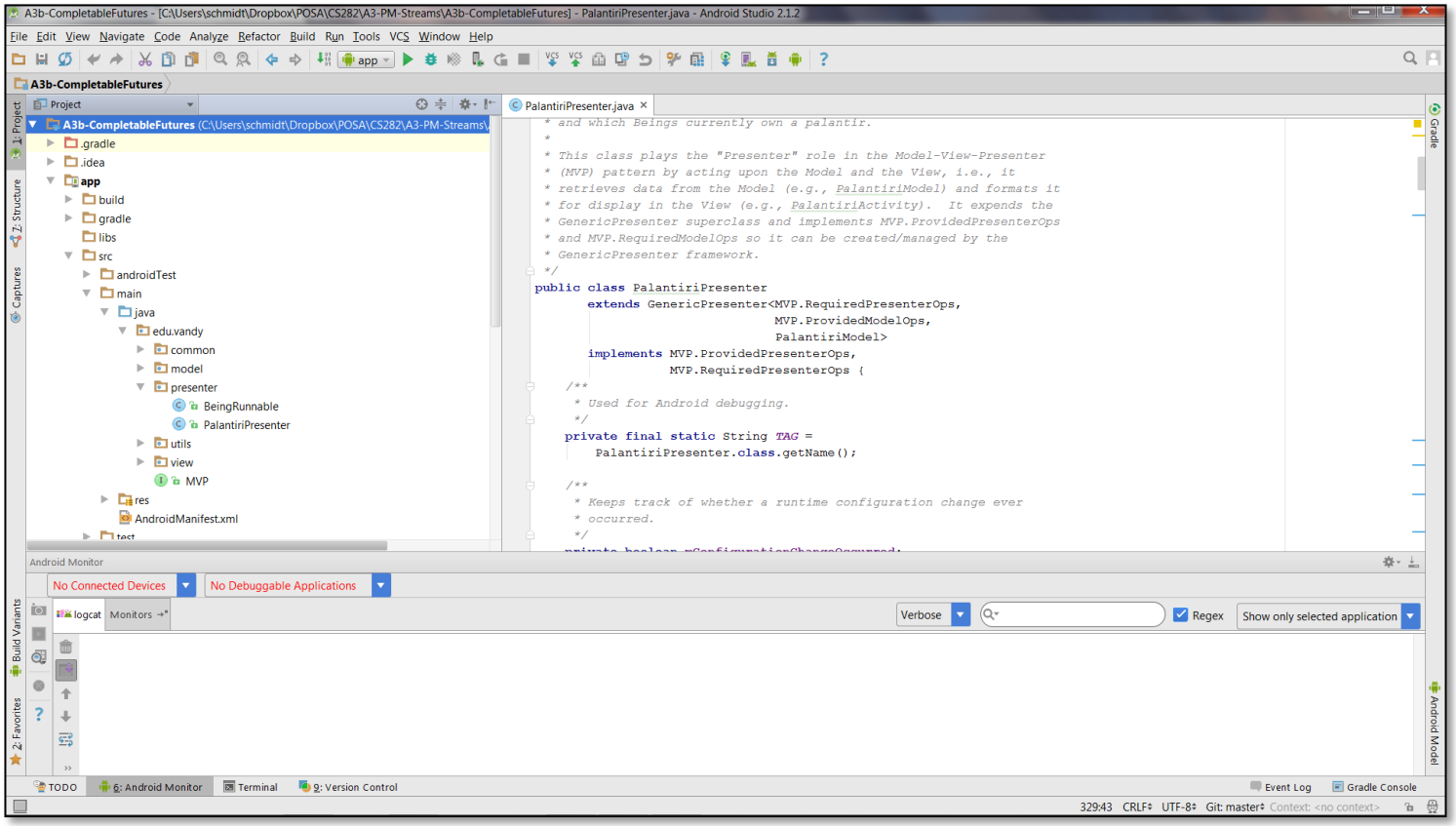
The Interpreter Pattern

Implementation in Java

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Interpreter* pattern can be applied to automate the processing of input expressions from users.
- Understand the structure and functionality of the *Interpreter* pattern.
- Know how to implement the *Interpreter* pattern in Java.



Douglas C. Schmidt

Implementing the Interpreter Pattern in Java

Interpreter example in Java

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

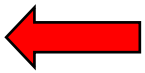
```
interface Expr {
    int interpret();
    ComponentNode build();
}

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(); }
    ...


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

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

Constructor initializes the field

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;
    }
}
```

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 super class for binary operator nonterminal expressions

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;
    }
}
```

Constructor initializes both of the fields

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 MultiplyExpr extends BinaryExpr {

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

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

Constructor initializes the super class fields

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 {


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

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

Interpret nonterminal by multiplying stored expressions

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

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

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

Parse input and build `parseTree`

```
    Expr parseTree = buildParseTree(inputExpression);

    if (!parseTree.isEmpty()) {

      optimizeParseTree(parseTree);

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

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);
    }
    ...
```

Override this hook method to optimize `parseTree`

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);
    }
    ...
```

Build `ExpressionTree` from `parseTree`

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.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.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();
  }
...
```

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

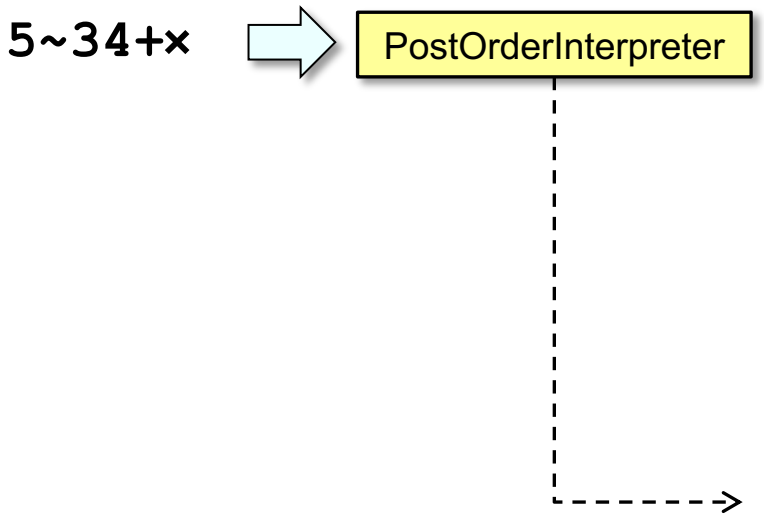
- `PostOrderInterpreter.buildParseTree()` returns the root of a parse tree corresponding to the input expression.

```
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();
  }
...
```

Pop top item off the stack, which contains the complete parse tree

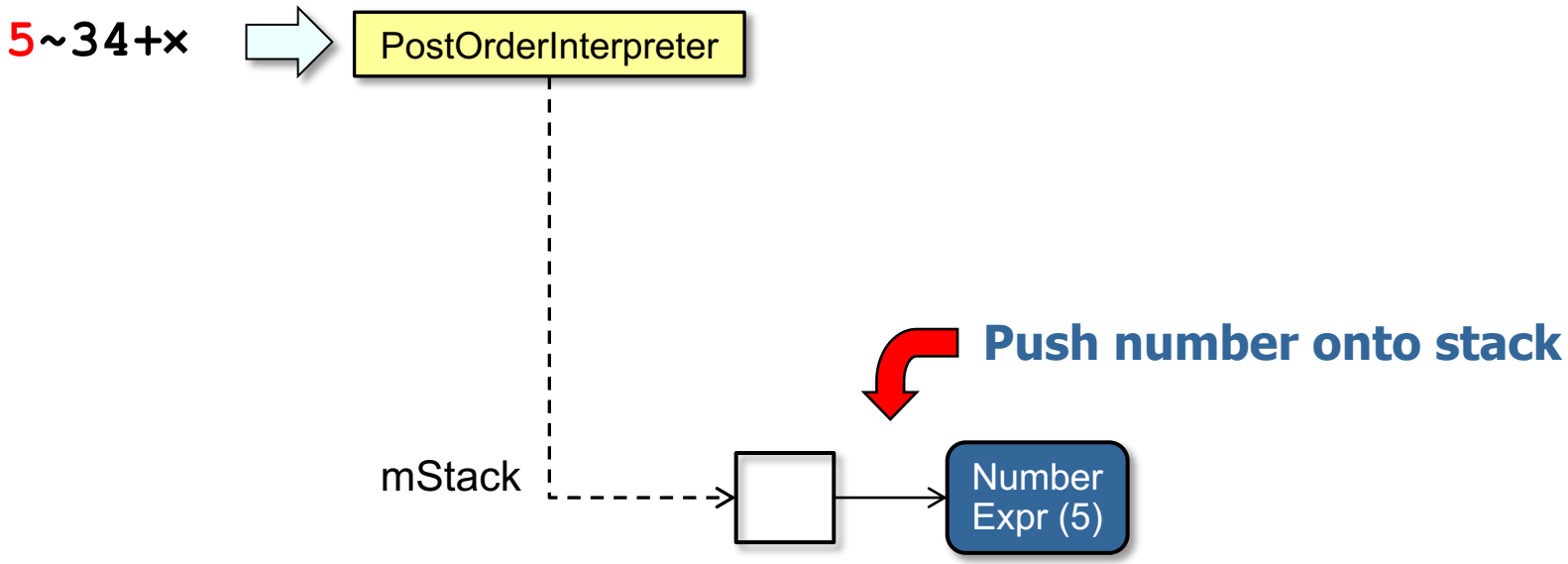
Interpreter example in Java

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



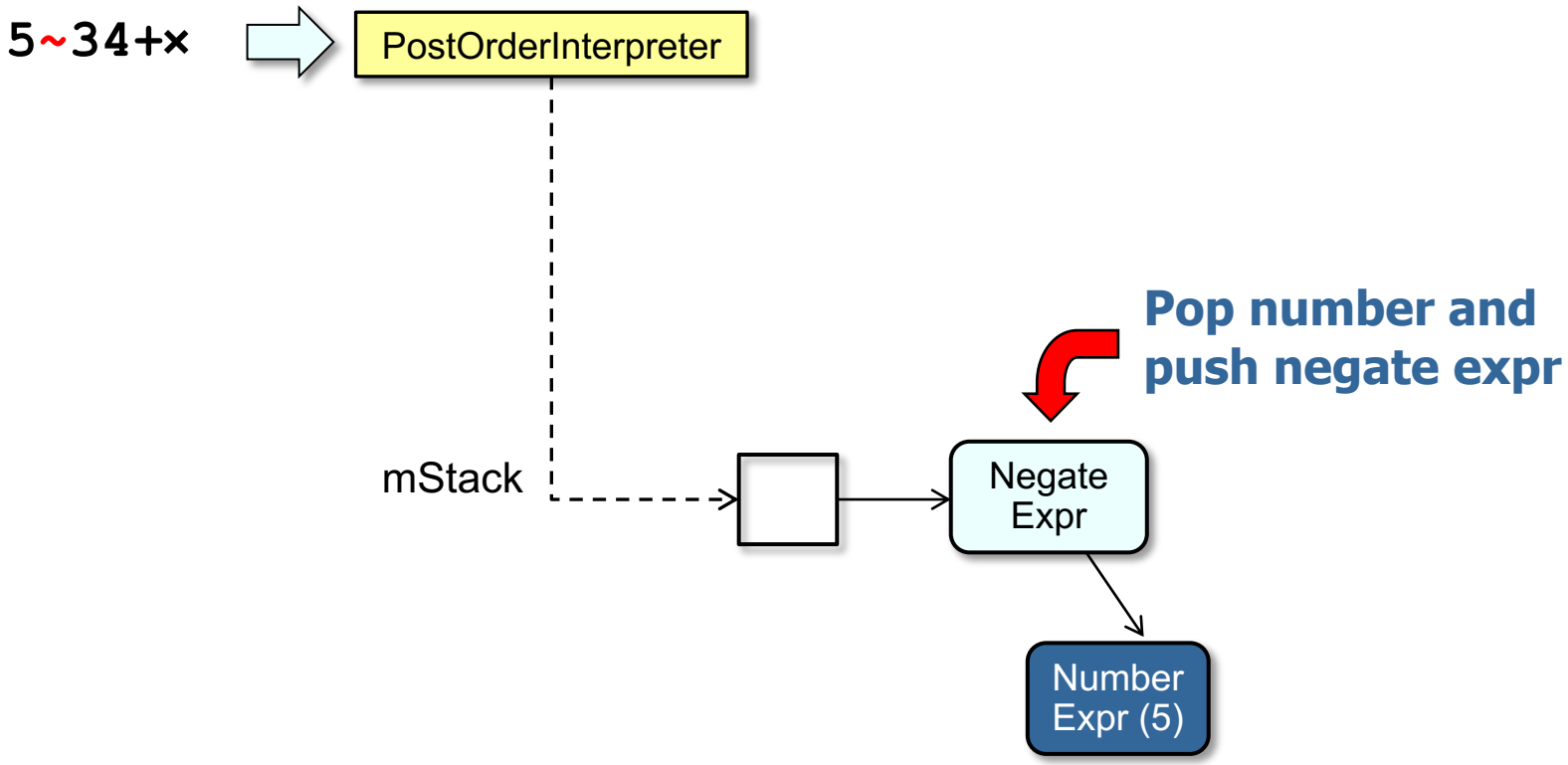
Interpreter example in Java

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



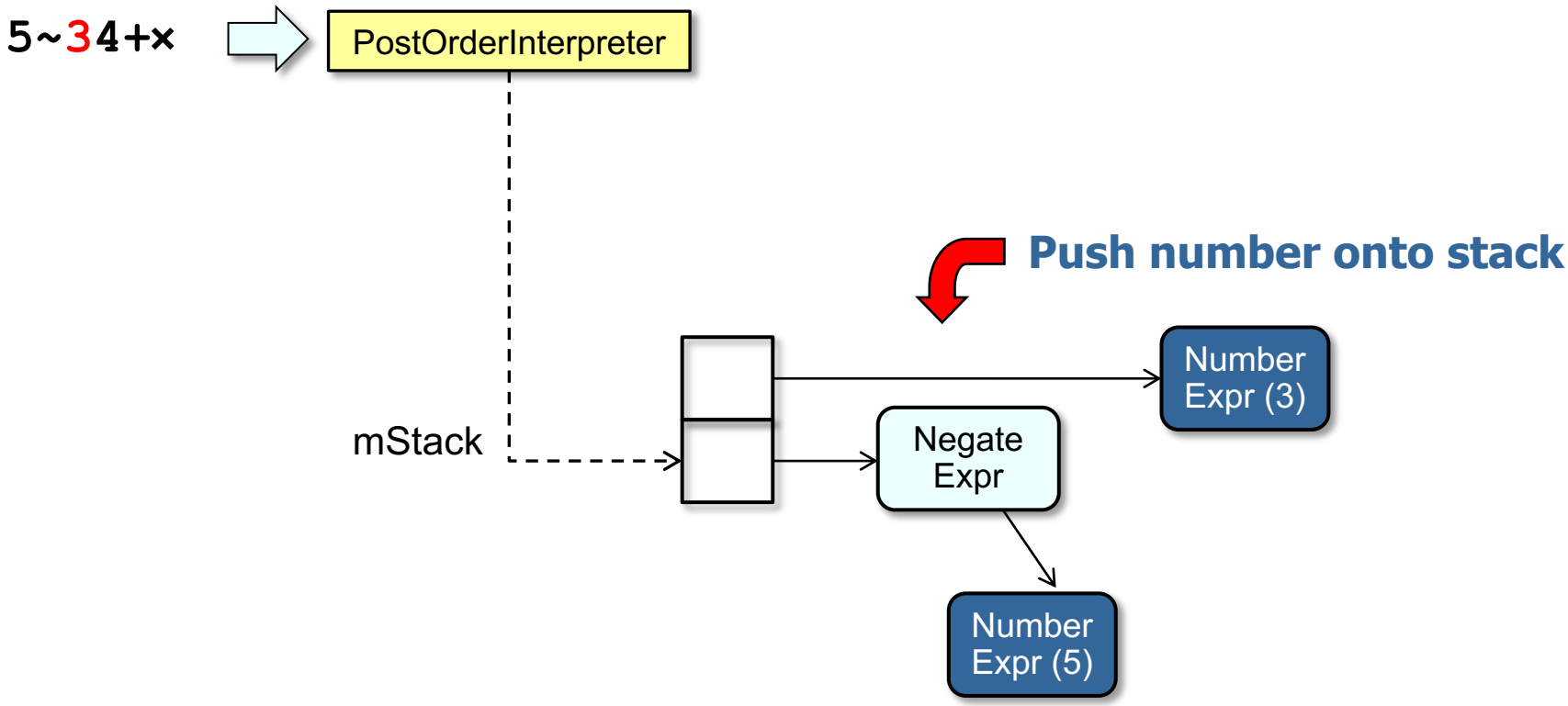
Interpreter example in Java

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



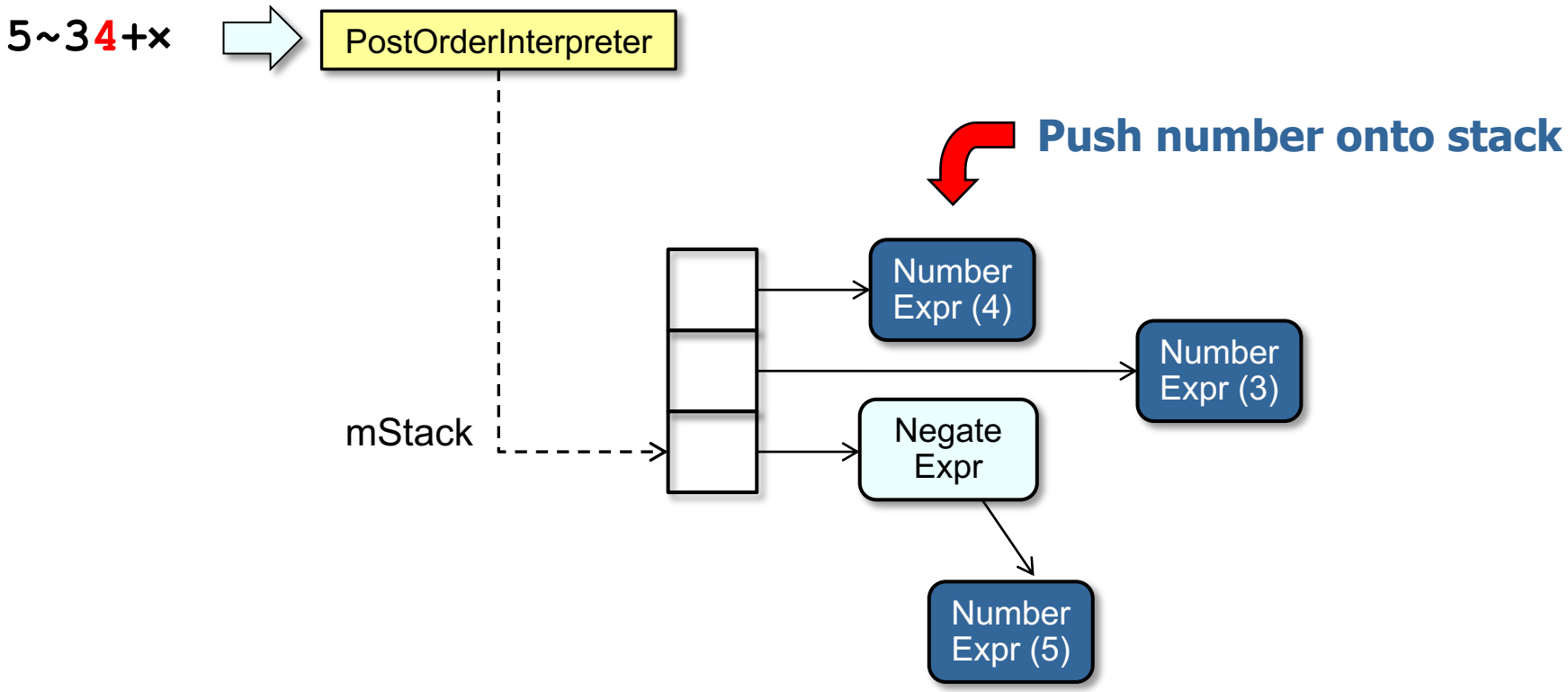
Interpreter example in Java

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



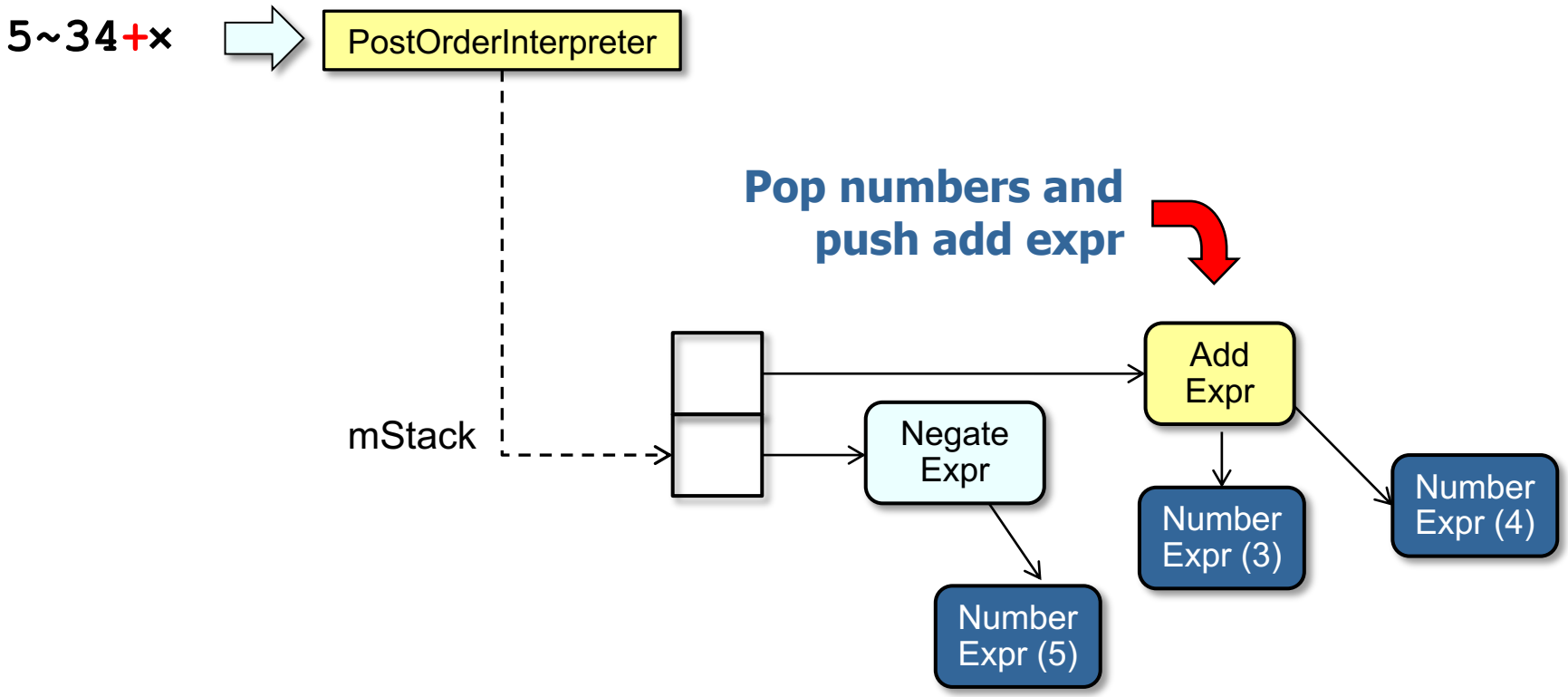
Interpreter example in Java

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



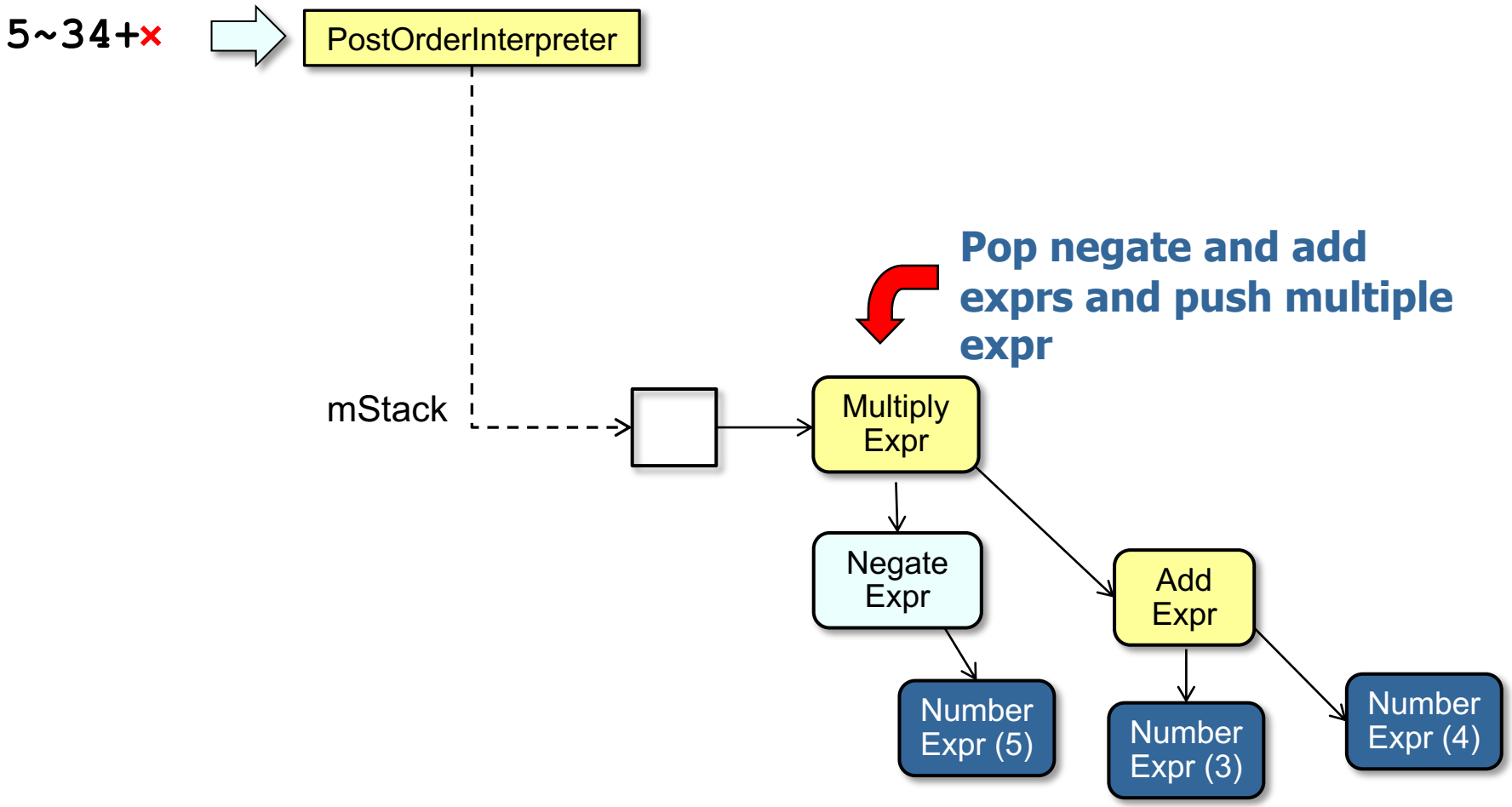
Interpreter example in Java

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



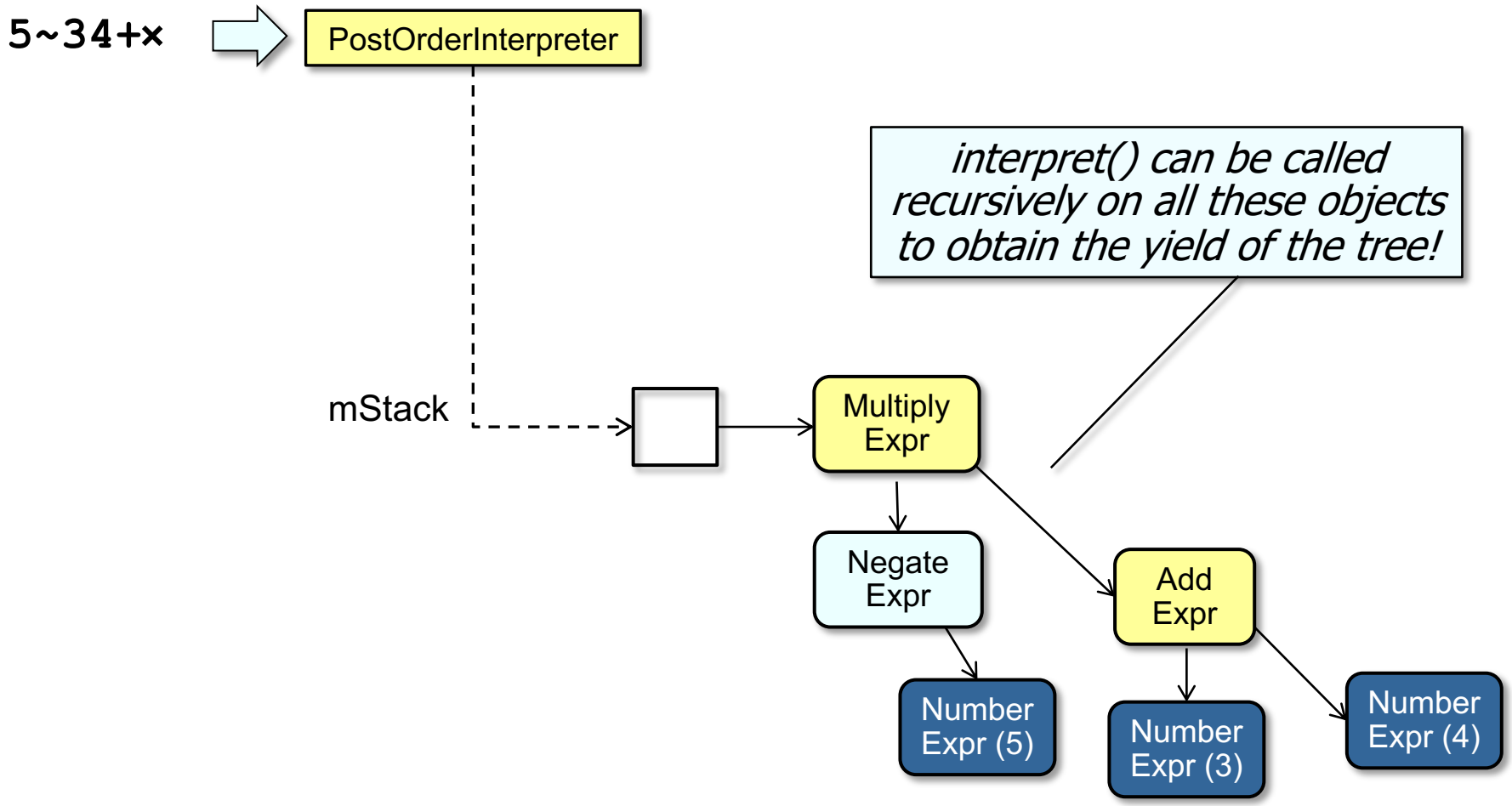
Interpreter example in Java

- **`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.



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) {
    if (mIndex >= mInputExpression.length())
      return false;
    else
      ...
```

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

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) {
    if (mIndex >= mInputExpression.length())
      return false;
    else
      ...
```

Bail out if we're at the end of the input

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

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++);


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

Get the next input character

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.

```
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++);


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

Handle numbers (terminal 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.

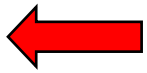
```
class ExprSpliterator
      extends Spliterators.AbstractSpliterator<Expr> {
  ...
  public boolean tryAdvance(Consumer<? super Expr> unUsed) {
    ...
    else { ...
      else {
        Expr rightExpr = mStack.pop();
        switch (c) {
          case '+':
             mStack.push(new AddExpr(mStack.pop(),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

Handle the addition operator

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 subtraction operator

The Interpreter Pattern

Other Considerations

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Interpreter* pattern can be applied to automate the processing of input expressions from users.
- Understand the structure and functionality of the *Interpreter* pattern.
- Know how to implement the *Interpreter* pattern in Java.
- Be aware of other considerations when applying the *Interpreter* pattern.

Douglas C. Schmidt

Other Considerations of the Interpreter Pattern

Consequences

+ Simple grammars are easy to change and extend


```
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.

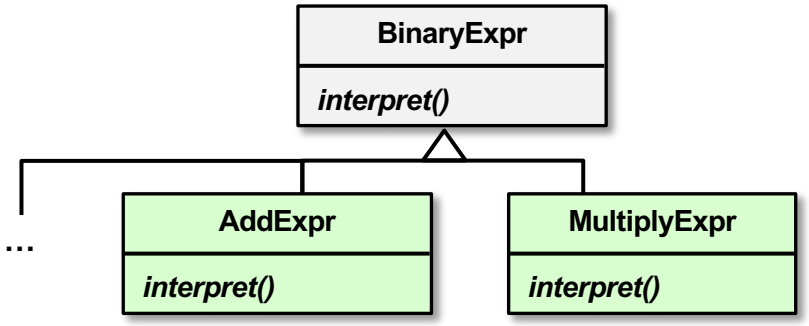
Consequences

+ Simple grammars are easy to change and extend, e.g.
  - All rules are 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 ')'
```

Consequences

+ Simple grammars are easy to change and extend, e.g.
  - All rules are represented by distinct classes in a consistent and orderly manner
  - Adding another rule simply 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

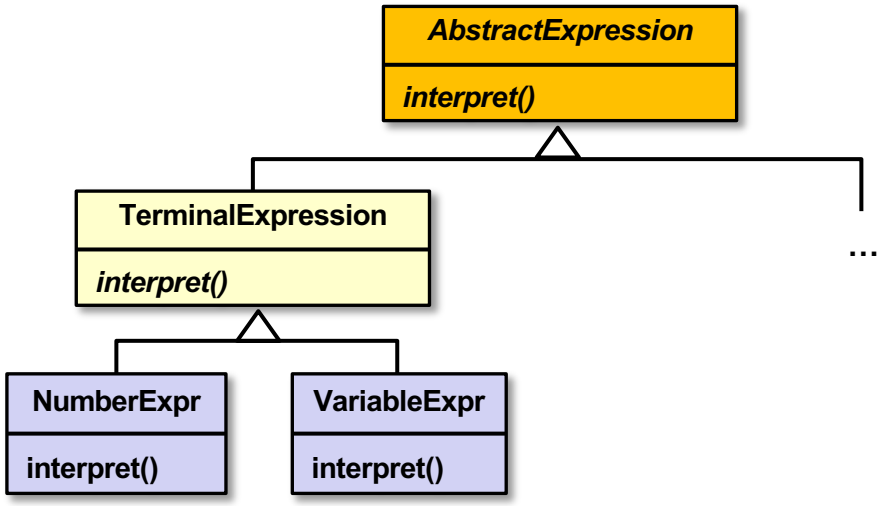
Complex grammars often require different approach, e.g., parser generators.

JavaCC™

See www.antlr.org and javacc.org

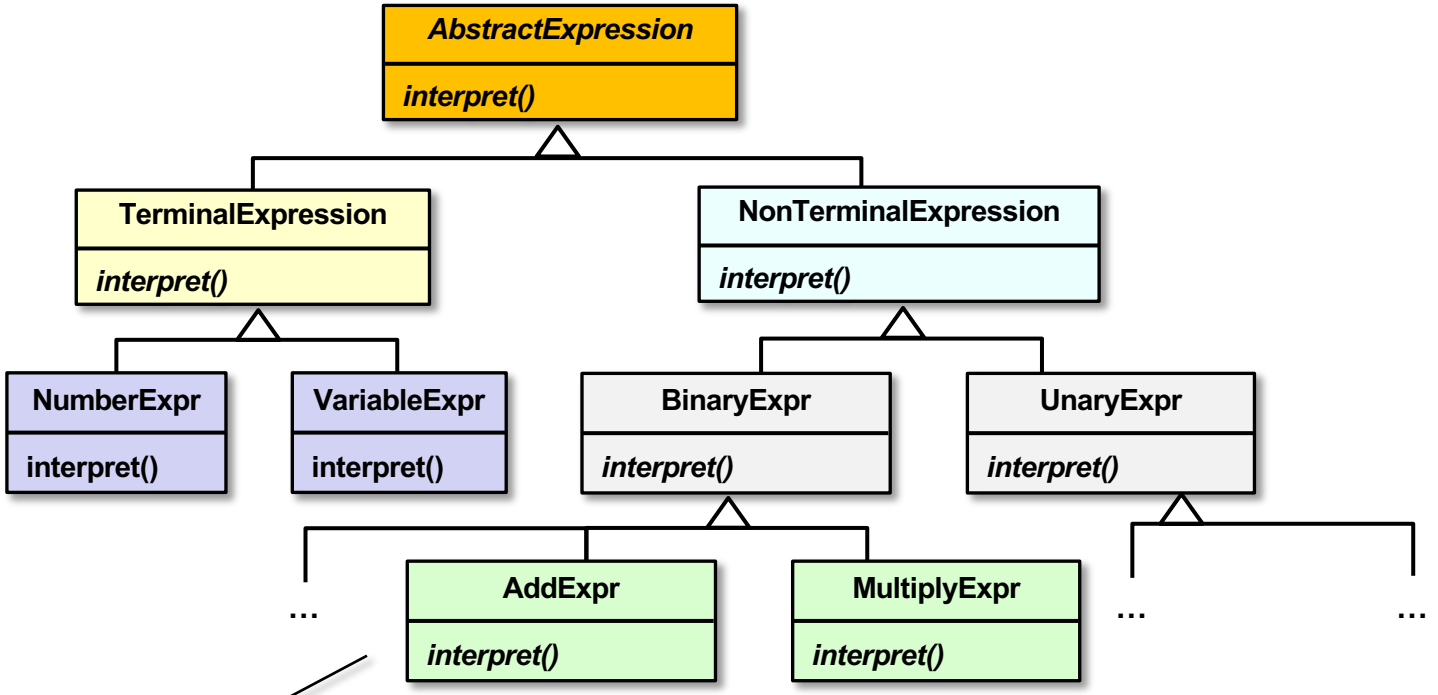
Implementation considerations

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



Implementation considerations

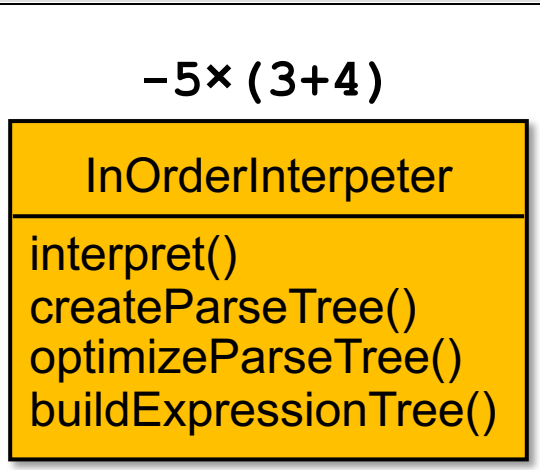
- Various approaches:
  - Express language rules, one per class.
    - Literal translations expressed as *terminal expressions*
    - Binary and unary nodes expressed as *nonterminal expressions*



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

Implementation considerations

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

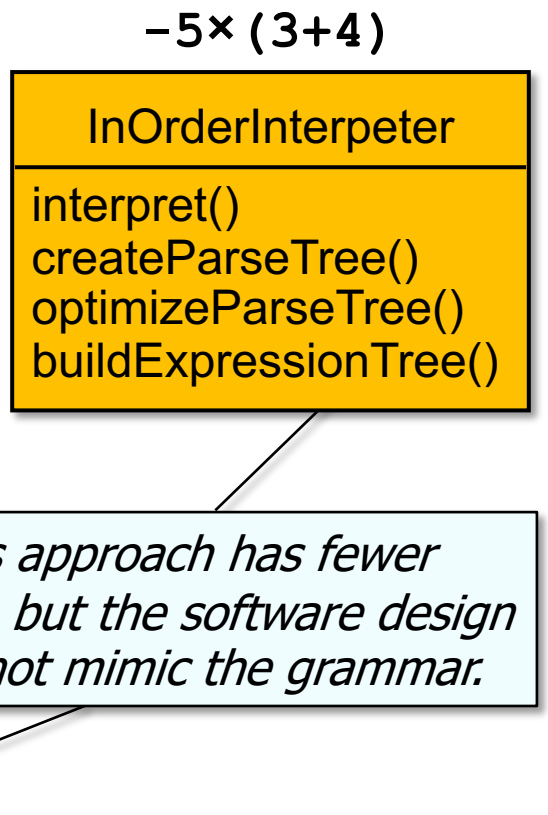



```
public final static int sMULTIPLICATION = 0;
public final static int sDIVISION = 1;
public final static int sADDITION = 2;
public final static int sSUBTRACTION = 3;
public final static int sNEGATION = 4;
public final static int sLPAREN = 5;
public final static int sRPAREN = 6;
public final static 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 considerations

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

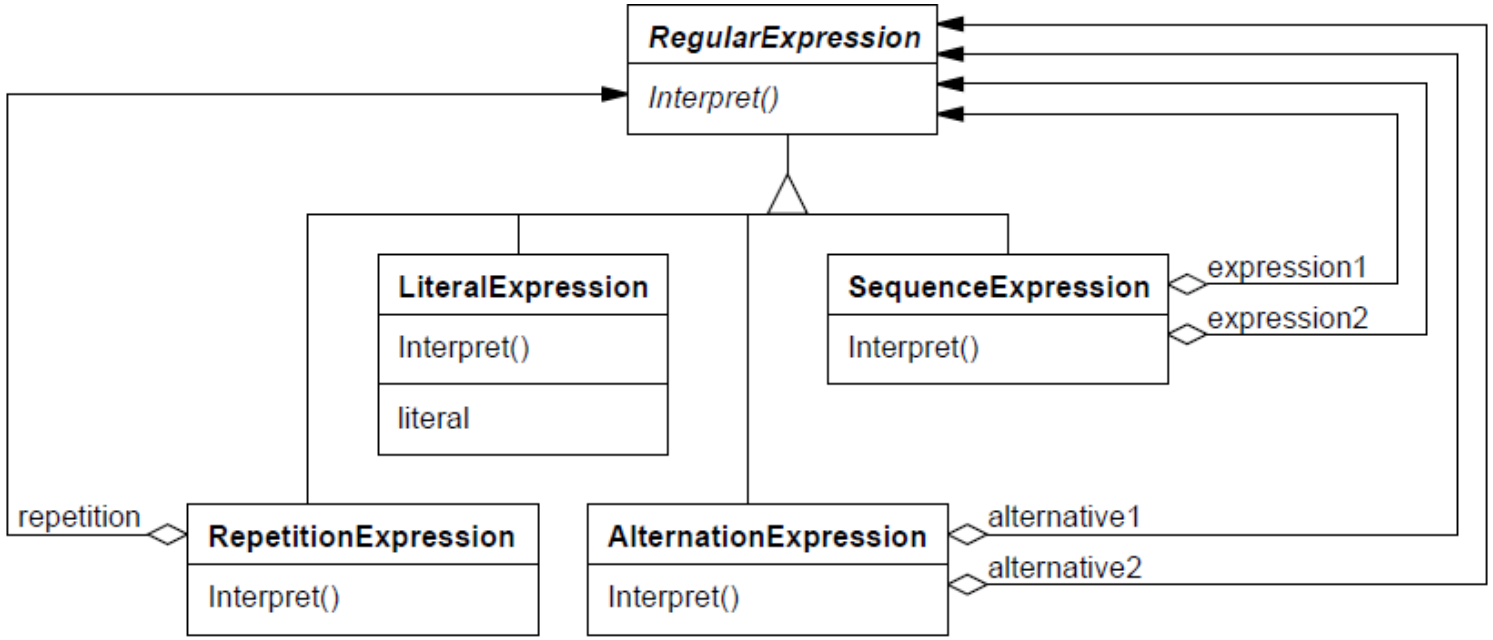



```
public final static int sMULTIPLICATION = 0;
public final static int sDIVISION = 1;
public final static int sADDITION = 2;
public final static int sSUBTRACTION = 3;
public final static int sNEGATION = 4;
public final static int sLPAREN = 5;
public final static int sRPAREN = 6;
public final static 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 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
```

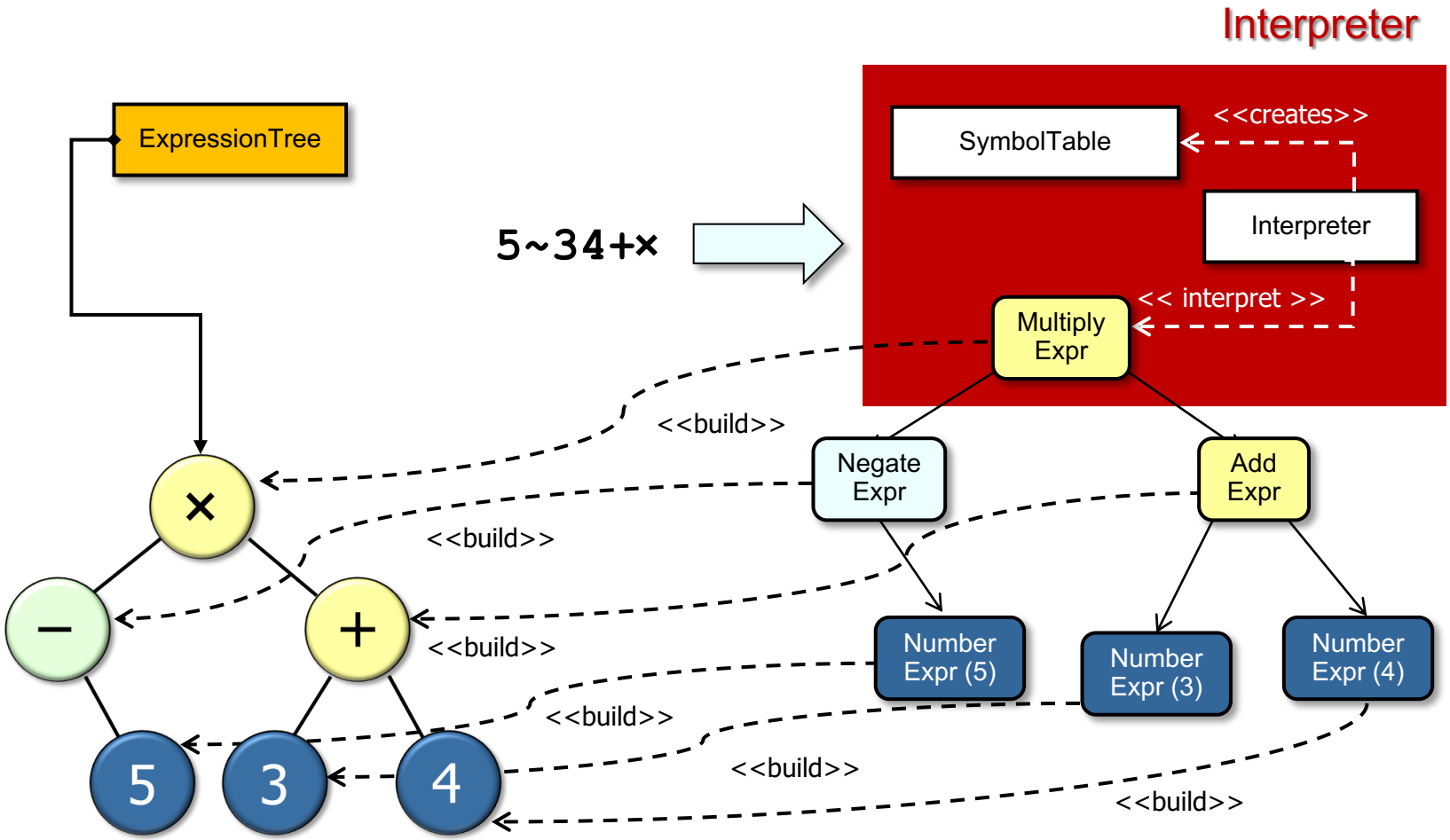
A compiled representation of a regular expression.

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.

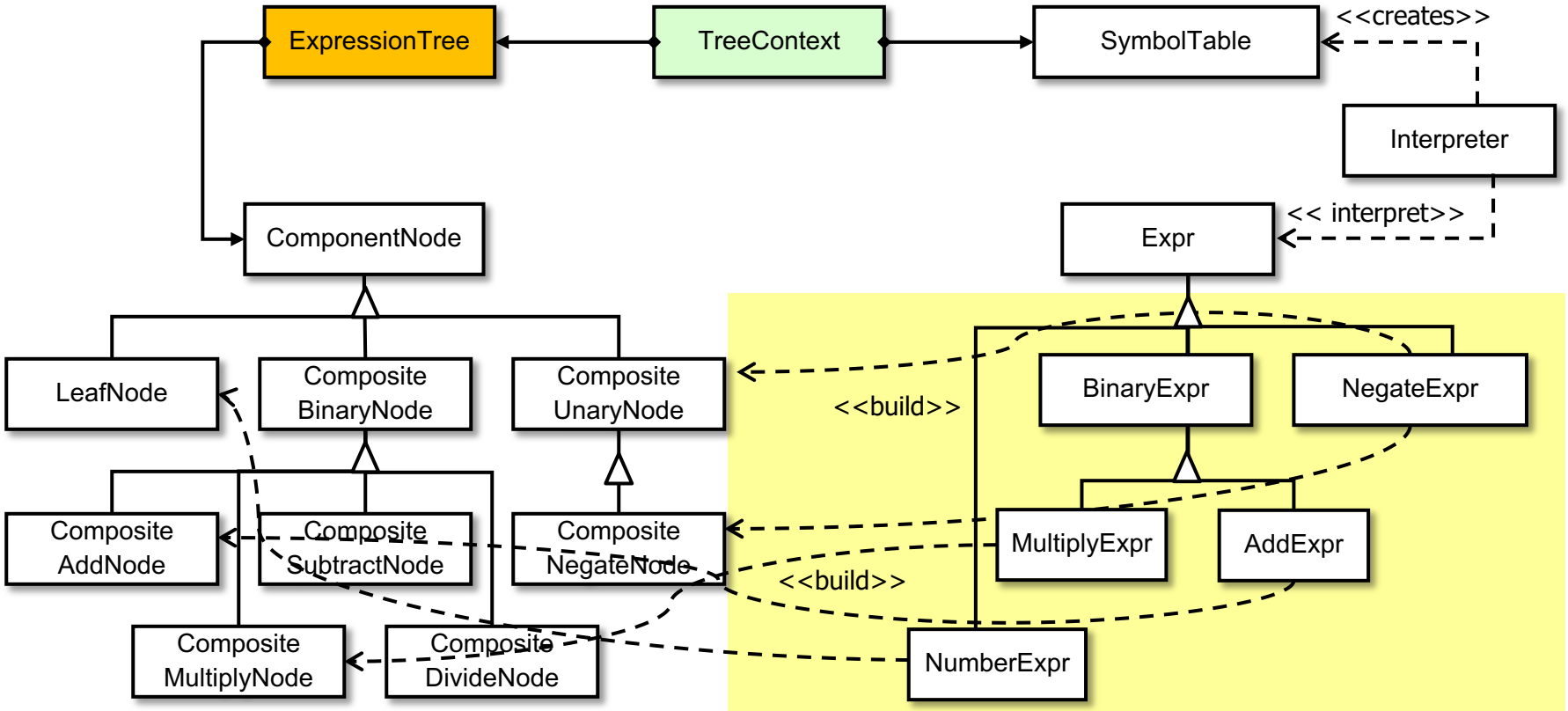
The Builder Pattern

Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

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

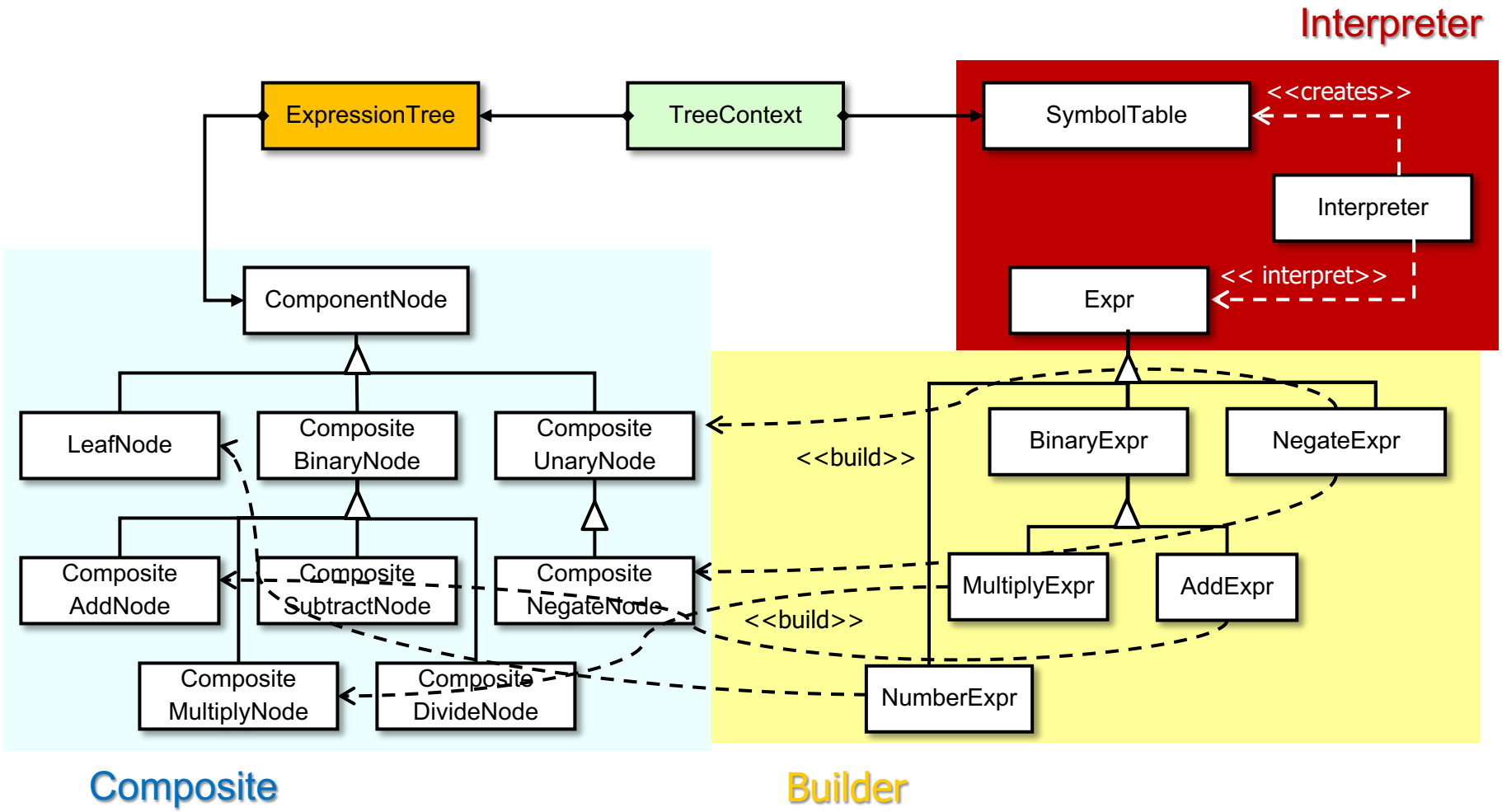


Douglas C. Schmidt

Motivating the Need for the Builder Pattern in the Expression Tree App

A Pattern for Building Objects Incrementally

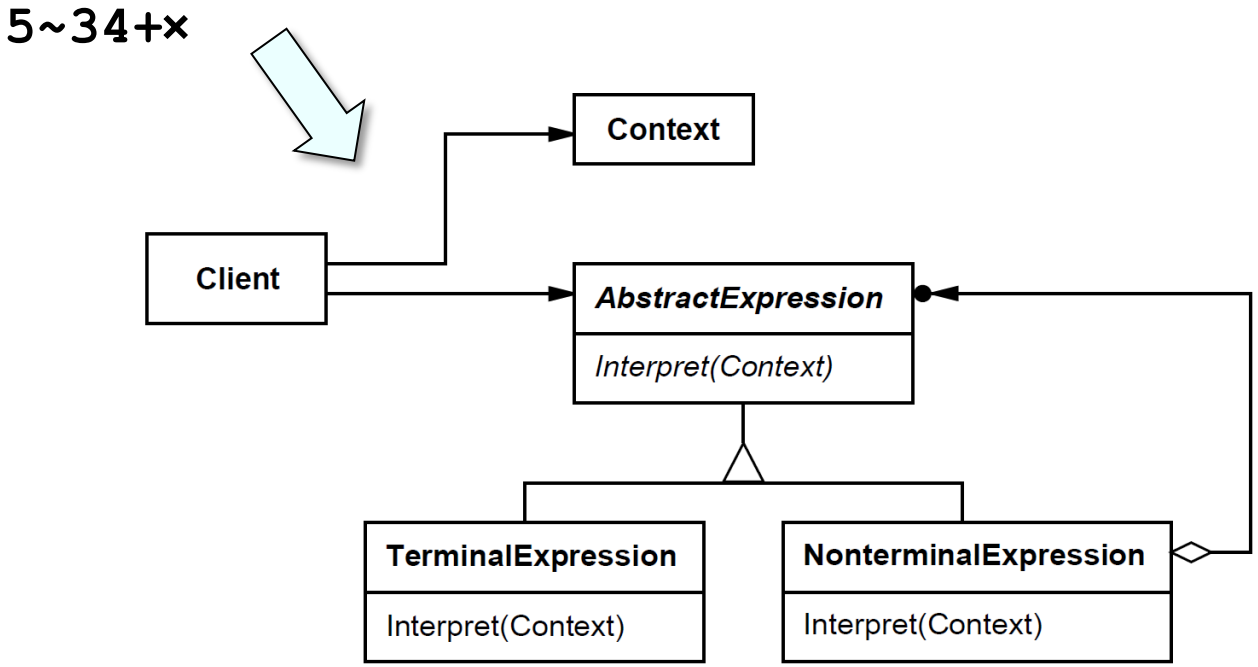
***Purpose**: Recursively builds the expression tree's Composite-based internal data structure from the Interpreter-generated parse tree*



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

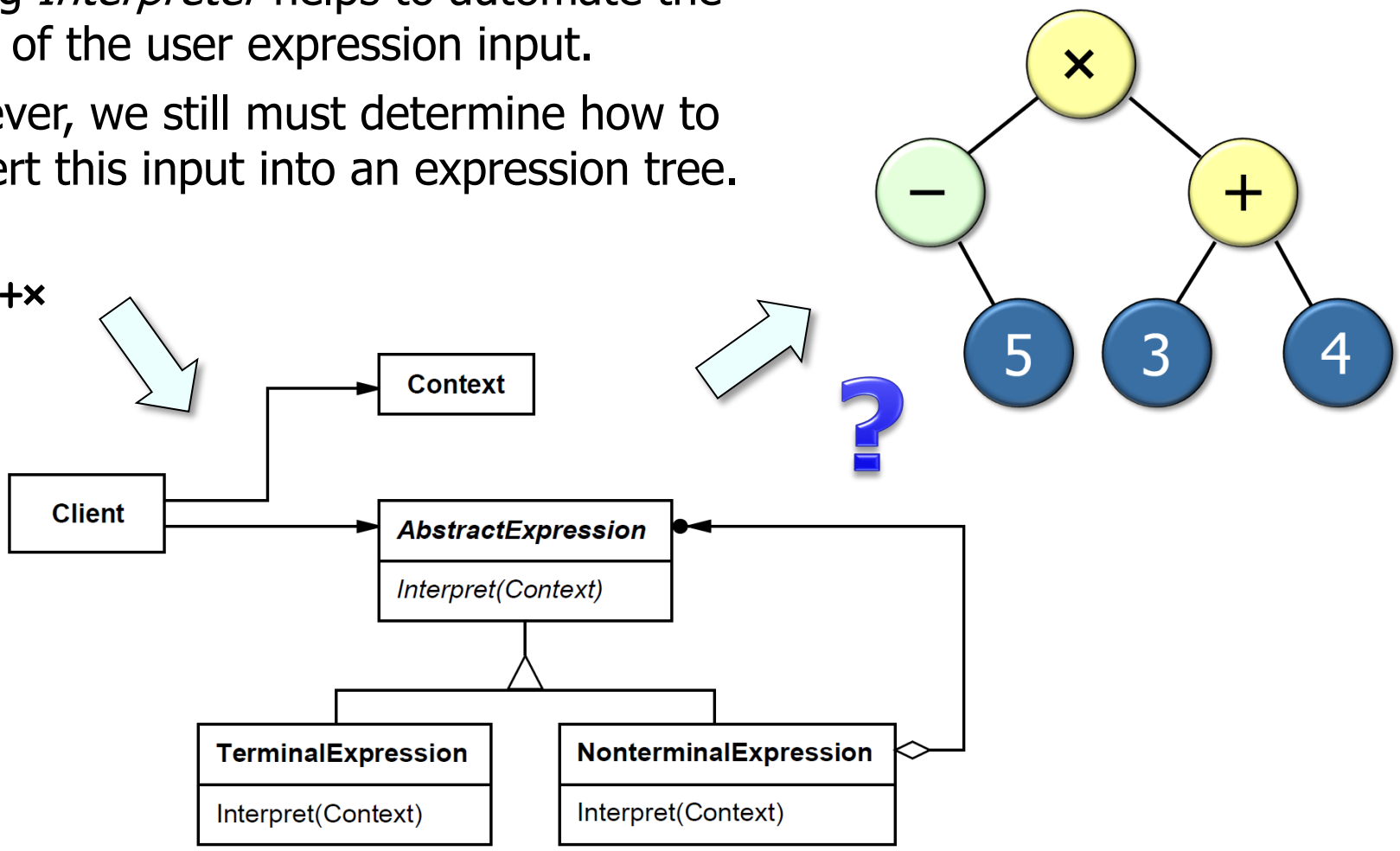
Context: OO Expression Tree Processing App

- Applying *Interpreter* helps to automate the parsing of the user expression input.



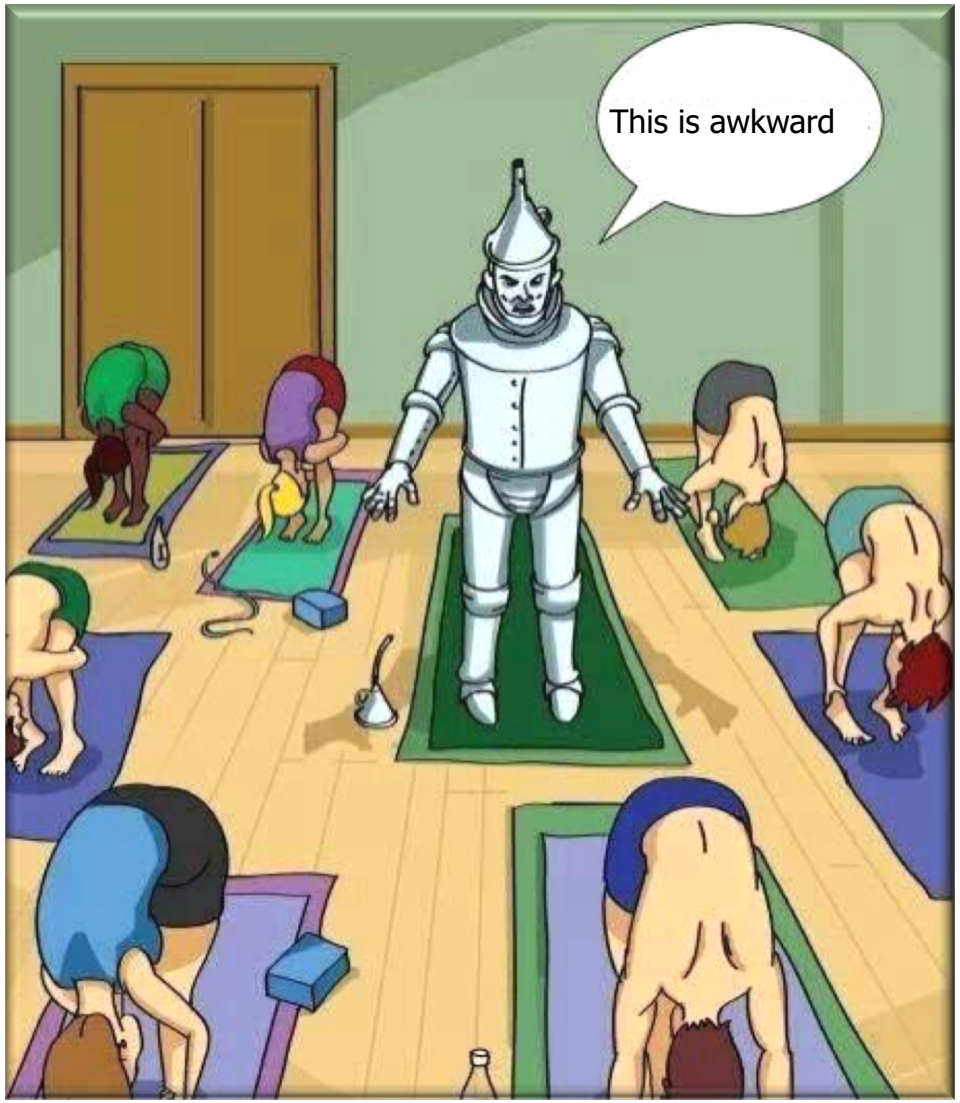
Context: OO Expression Tree Processing App

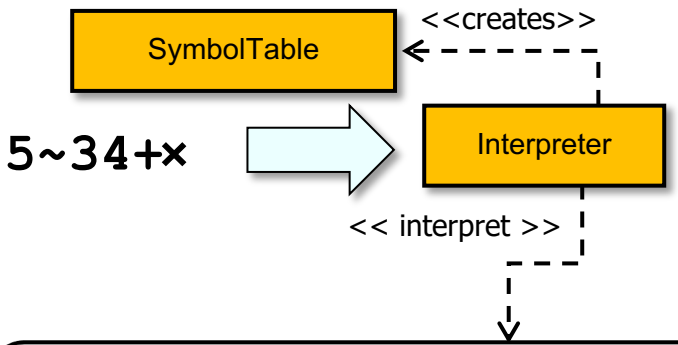
- Applying *Interpreter* helps to automate the parsing of the user expression input.
  - However, we still must determine how to convert this input into an expression tree.



Problem: Inflexible Interpreter Output

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

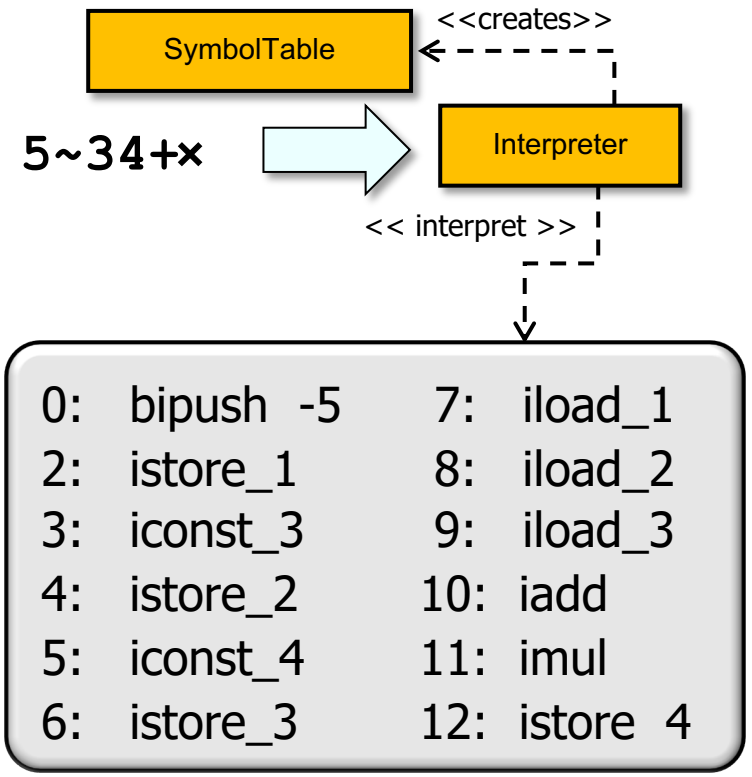





```
0:  bipush  -5      7:  iload_1
2:  istore_1        8:  iload_2
3:  iconst_3        9:  iload_3
4:  istore_2       10:  iadd
5:  iconst_4       11:  imul
6:  istore_3       12:  istore  4
```

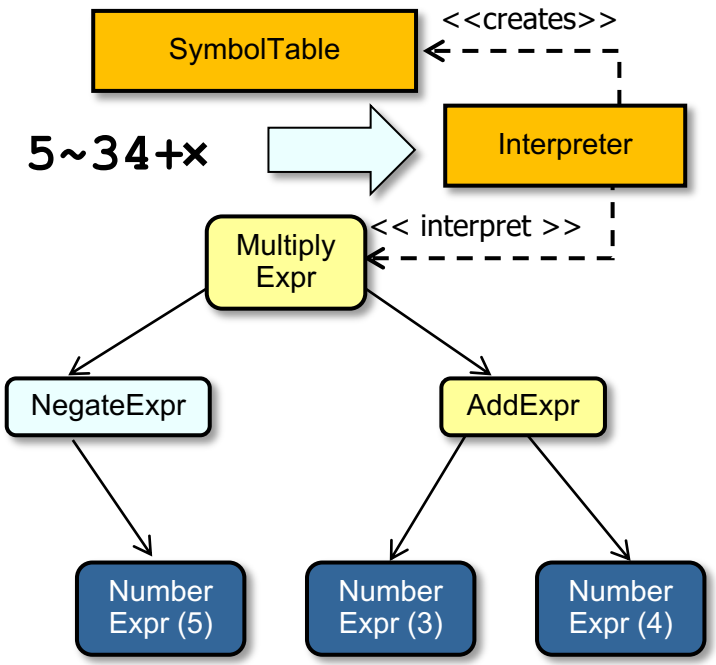
Problem: Inflexible Interpreter Output

- Hard-coding **`Interpreter`** to only generate one type of output is inflexible.
  - e.g., it precludes additional semantic analysis and optimization.



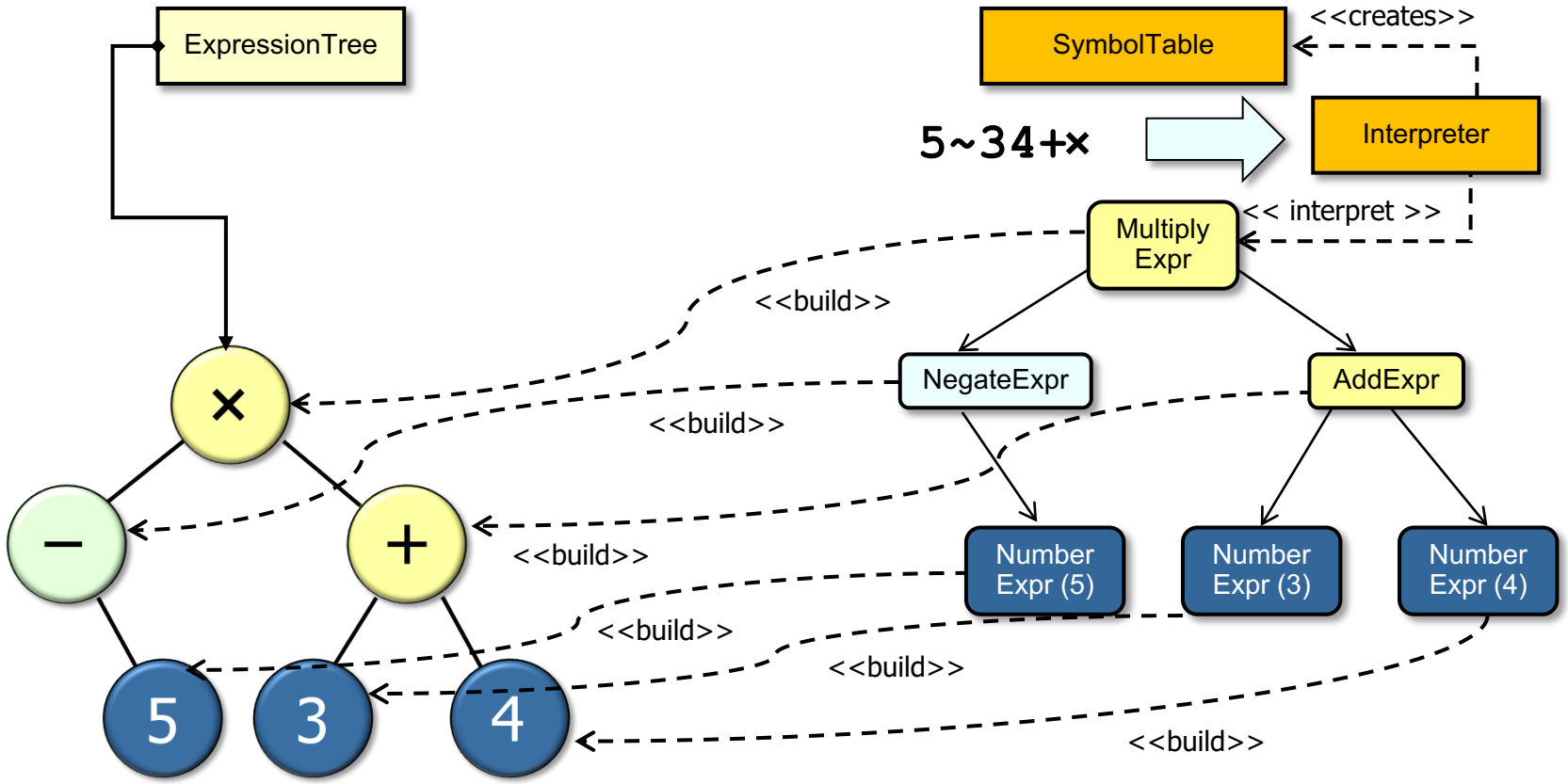
Solution: Build Complex Object Incrementally

- Have **`Interpreter`** create a parse tree.
  - Optionally, analyze and optimize this parse tree during a subsequent processing phase.



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)
```

<<creates>>

```
                Expr(Expr left,
                     Expr right,
                     int precedence)
           int  precedence()
 ComponentNode  build()
```

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)
```

Abstract super class of all parse tree nodes

<<creates>>

```
Expr(Expr left,
     Expr right,
     int precedence)
   int precedence()
ComponentNode build()
```

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)
```

<<creates>>

```
                Expr(Expr left,
                     Expr right,
                     int precedence)
           int  precedence()
ComponentNode   build()
```

This method builds a component node corresponding to the parse tree node

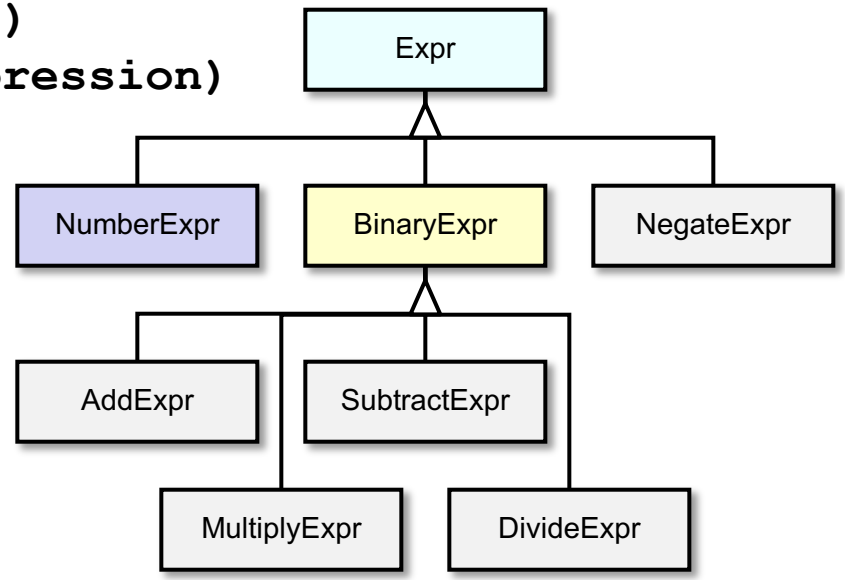
Expr Class Hierarchy Overview

- 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)
```

Interpreter creates the parse tree incrementally by instantiating `Expr` subclasses

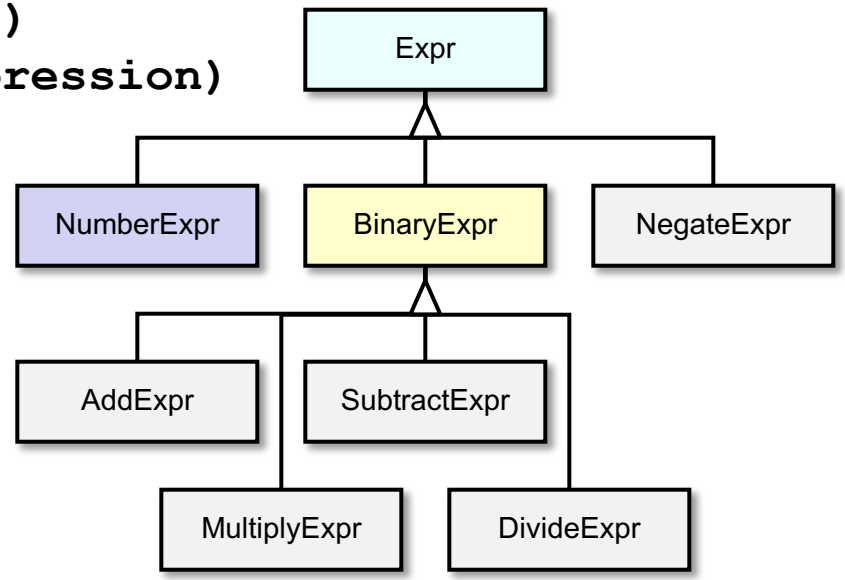


Expr Class Hierarchy Overview

- 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

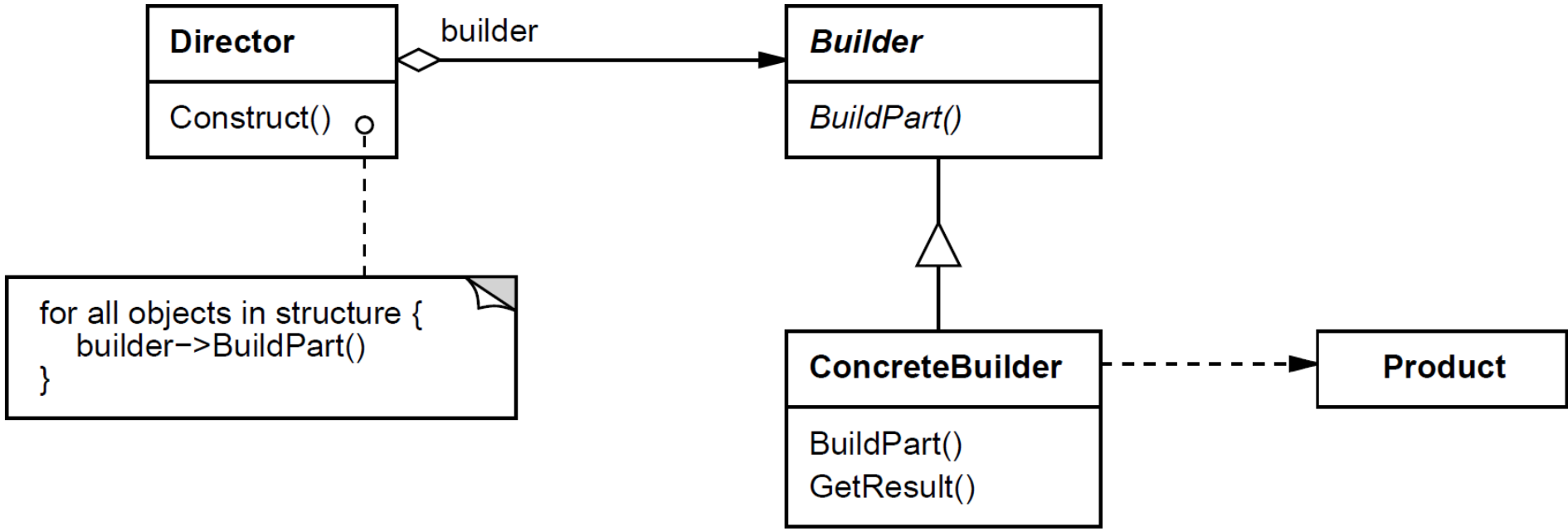
The Builder Pattern

Structure and Functionality

Douglas C. Schmidt

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.



Douglas C. Schmidt

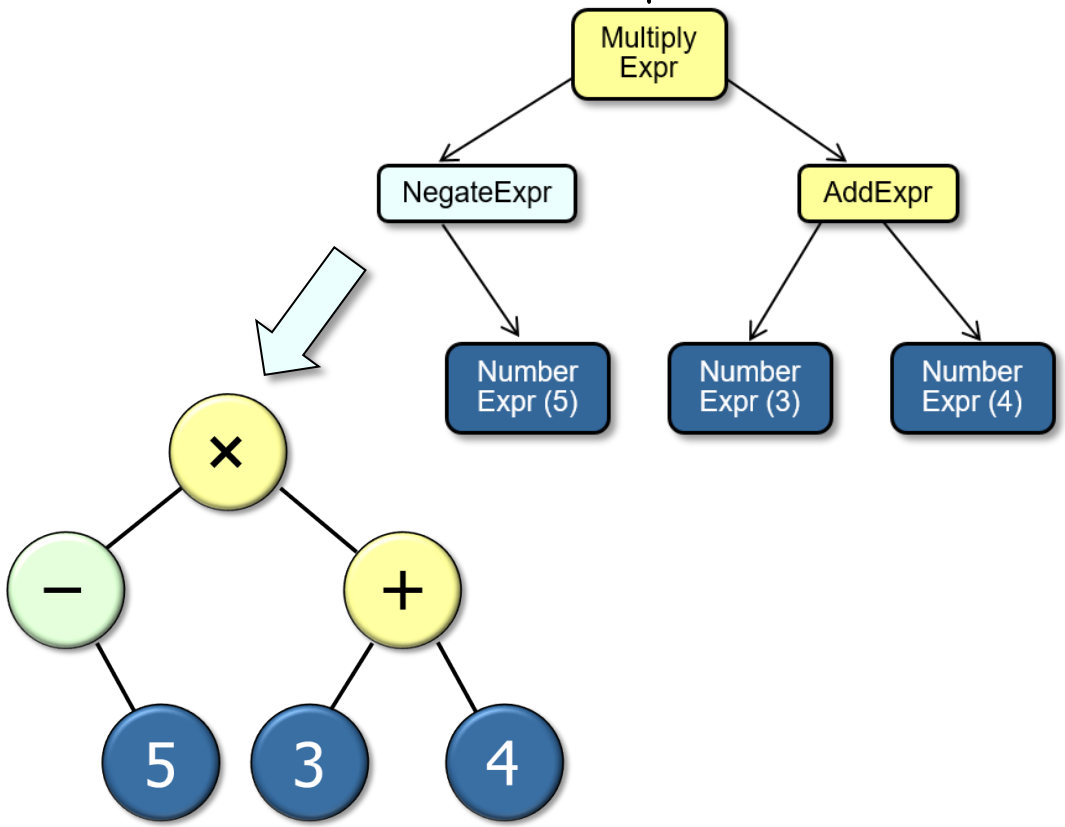
Structure and Functionality of the Builder Pattern

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

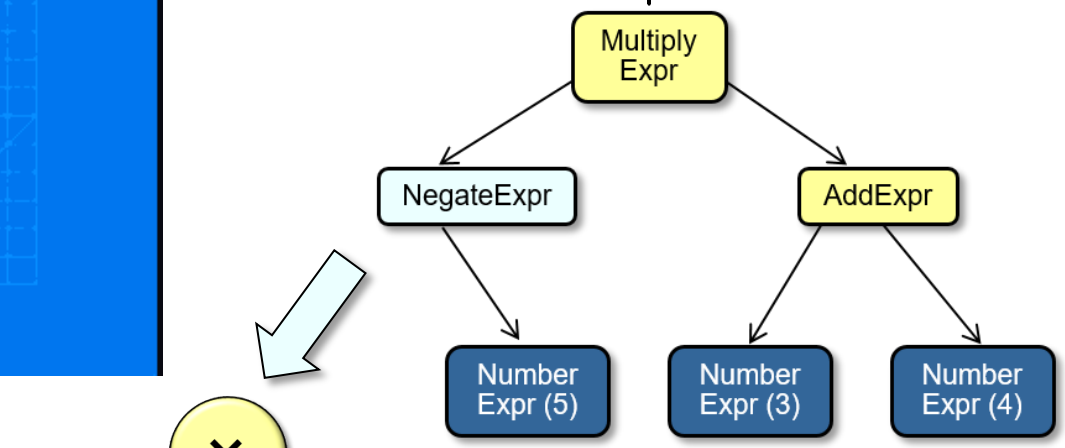
- Need to isolate the knowledge of creating a complex object from its parts

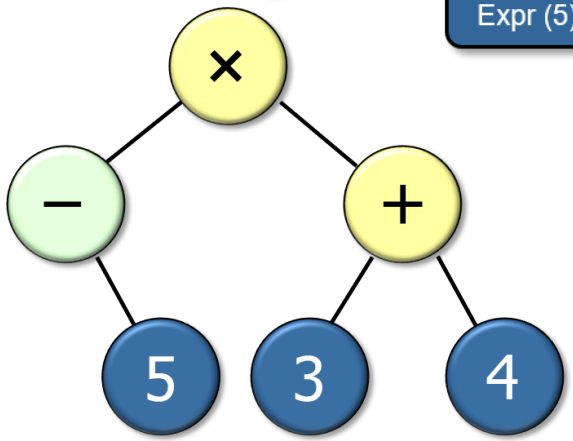
"Pre-order" input = `×-5+34`

"Post-order" input = `5-34+×`

"Level-order" input = `×-+534`

"In-order" input = `-5×(3+4)`

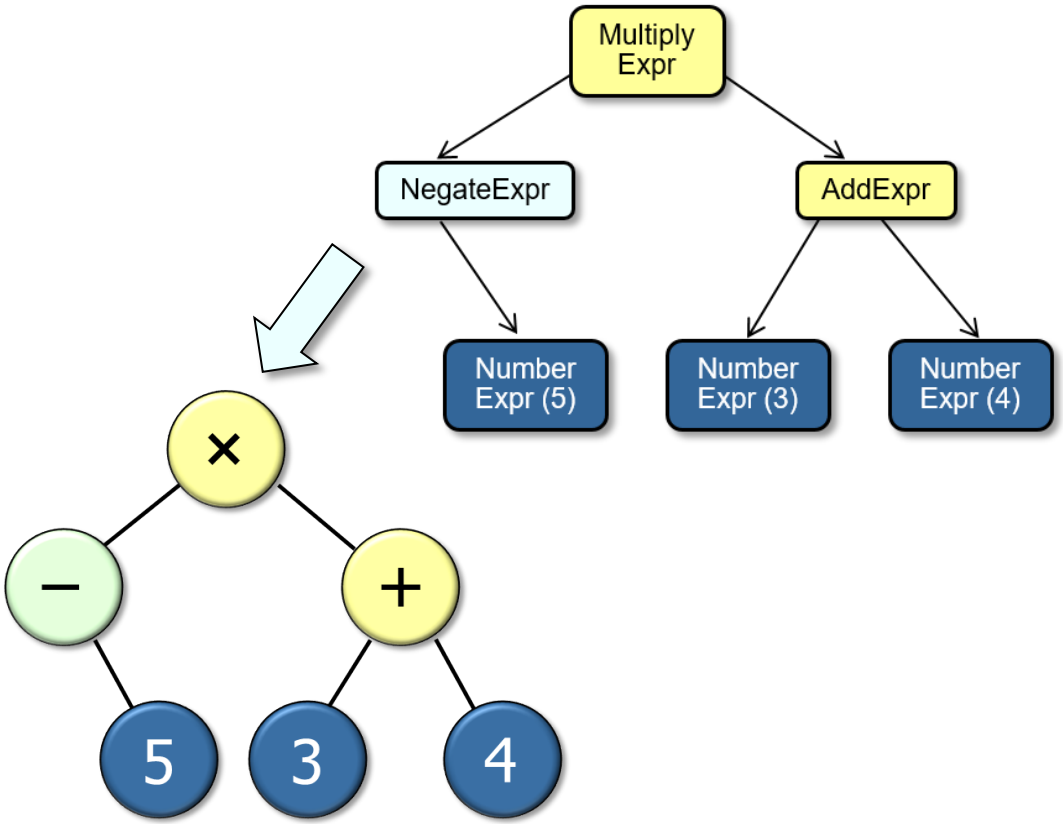




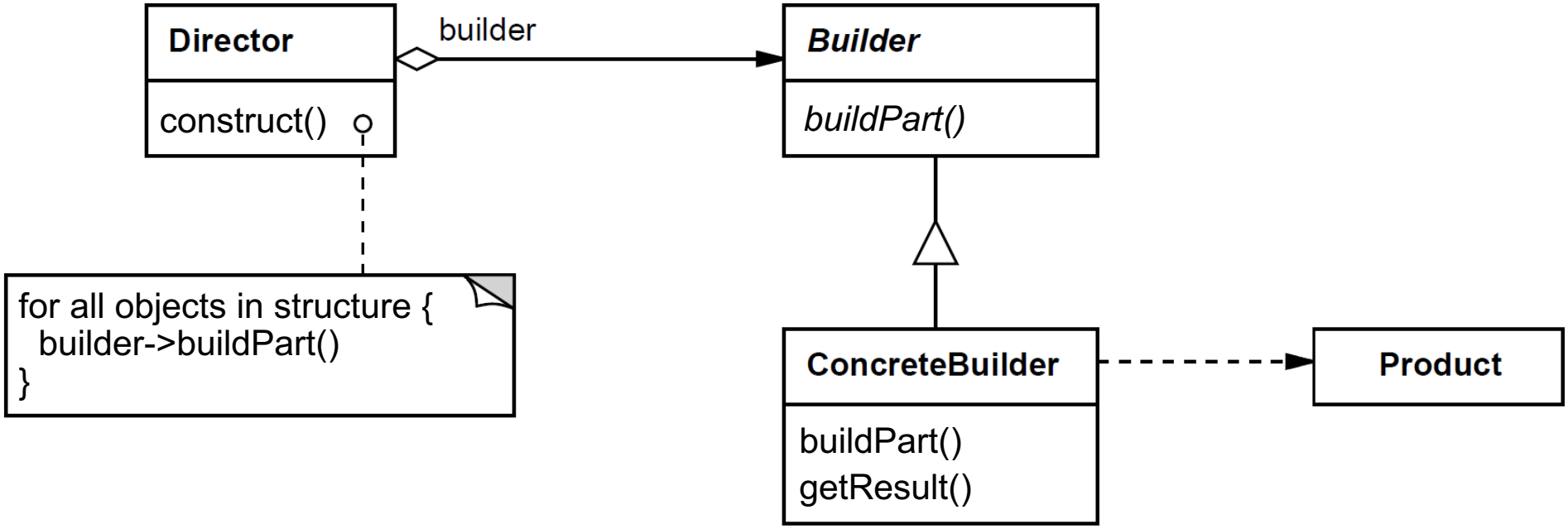
Applicability

- 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 parts

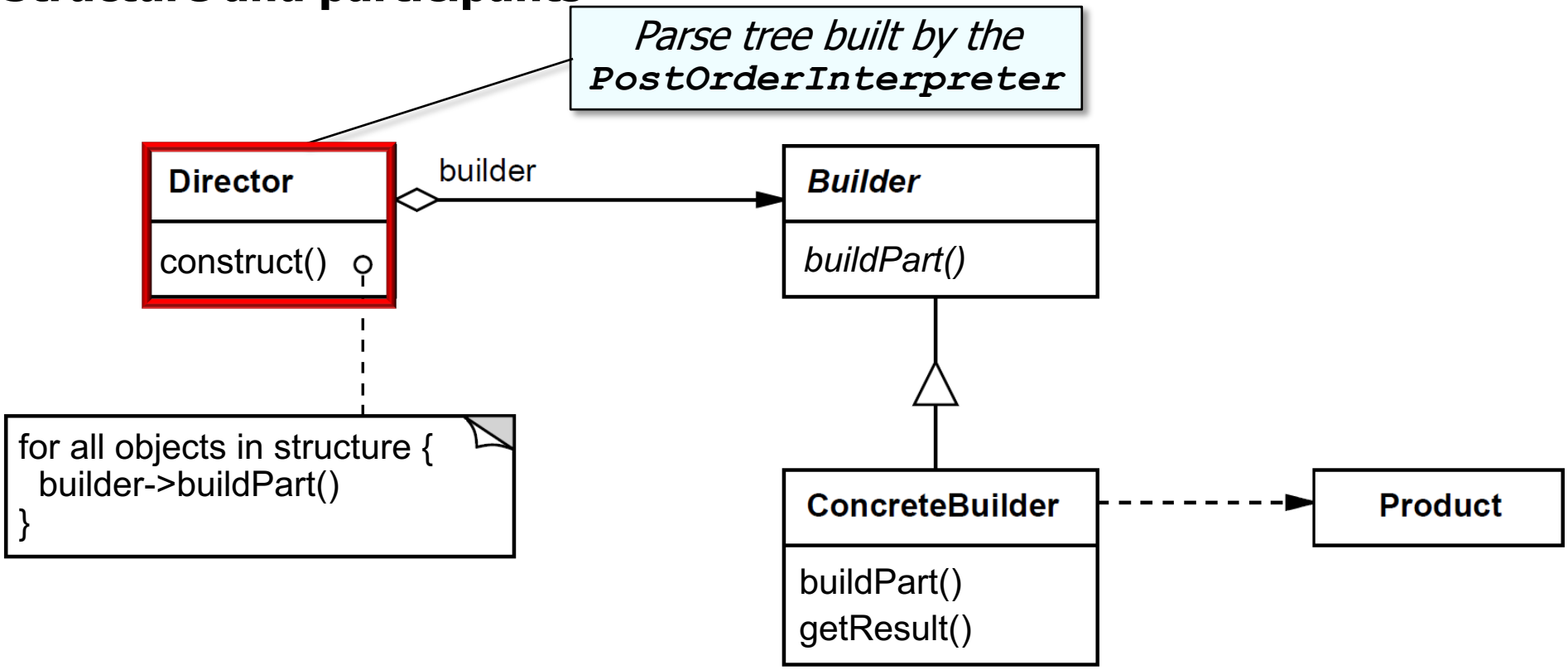
"Pre-order" input = `×-5+34`
"Post-order" input = `5-34+×`
"Level-order" input = `×-+534`
"In-order" input = `-5×(3+4)`



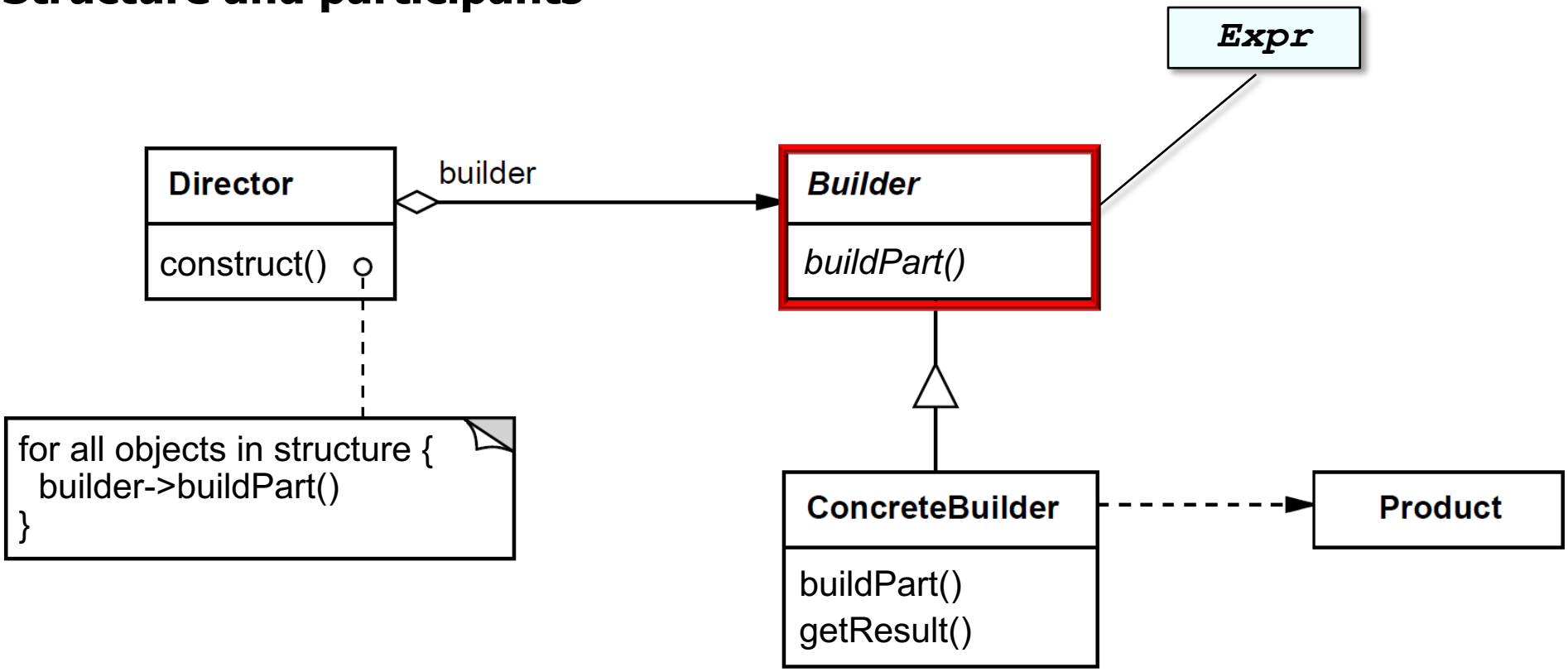
Structure and participants



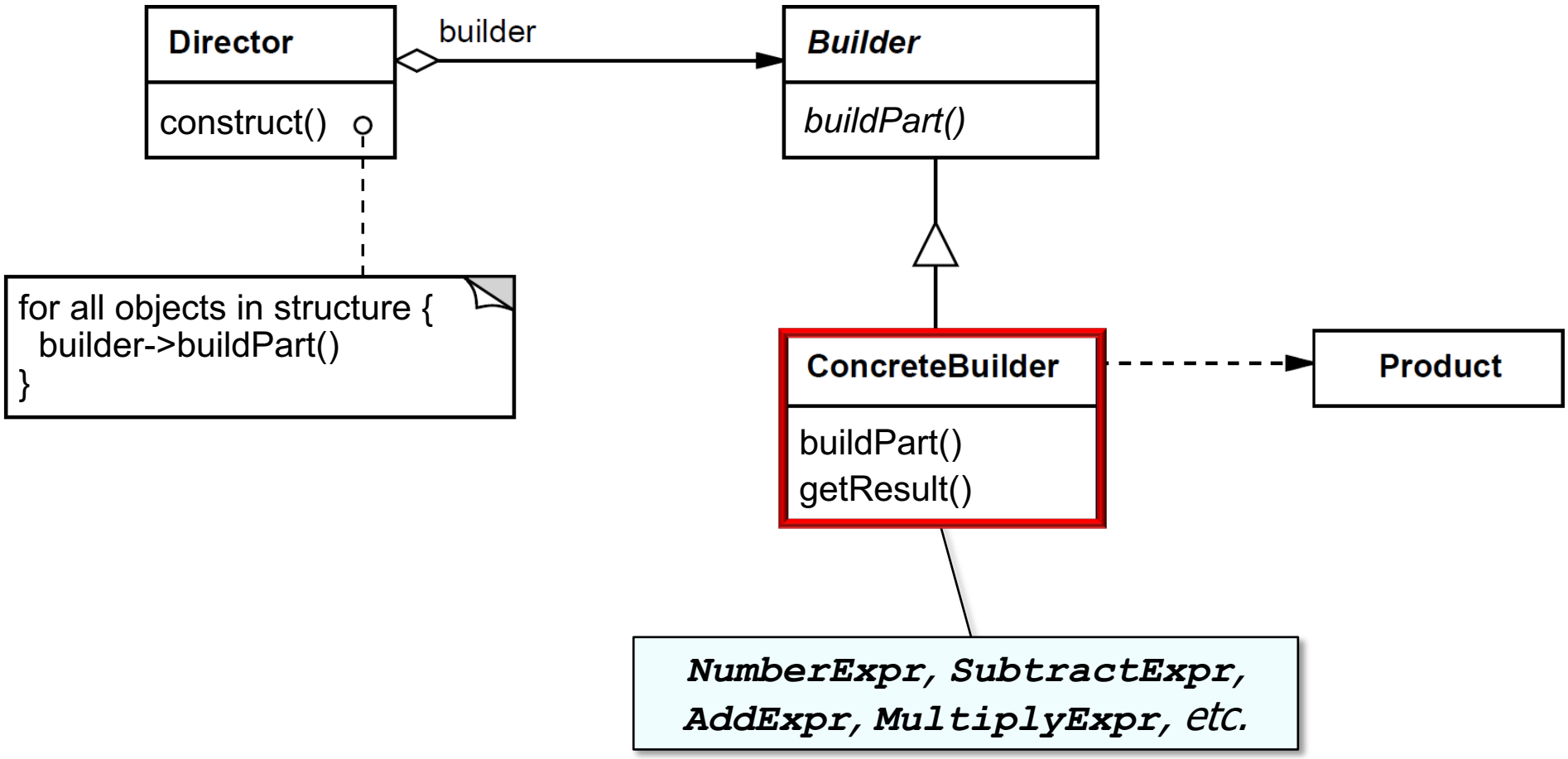
Structure and participants



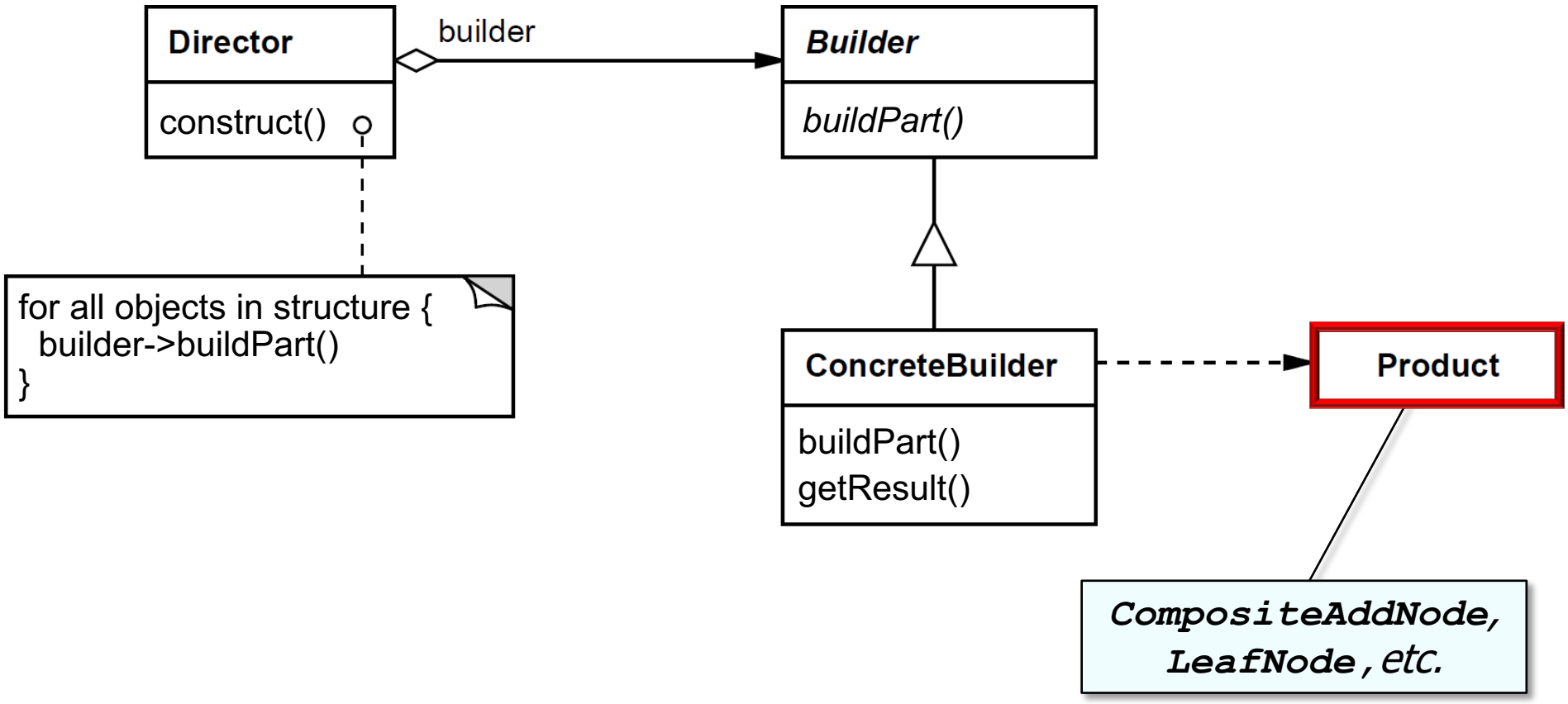
Structure and participants



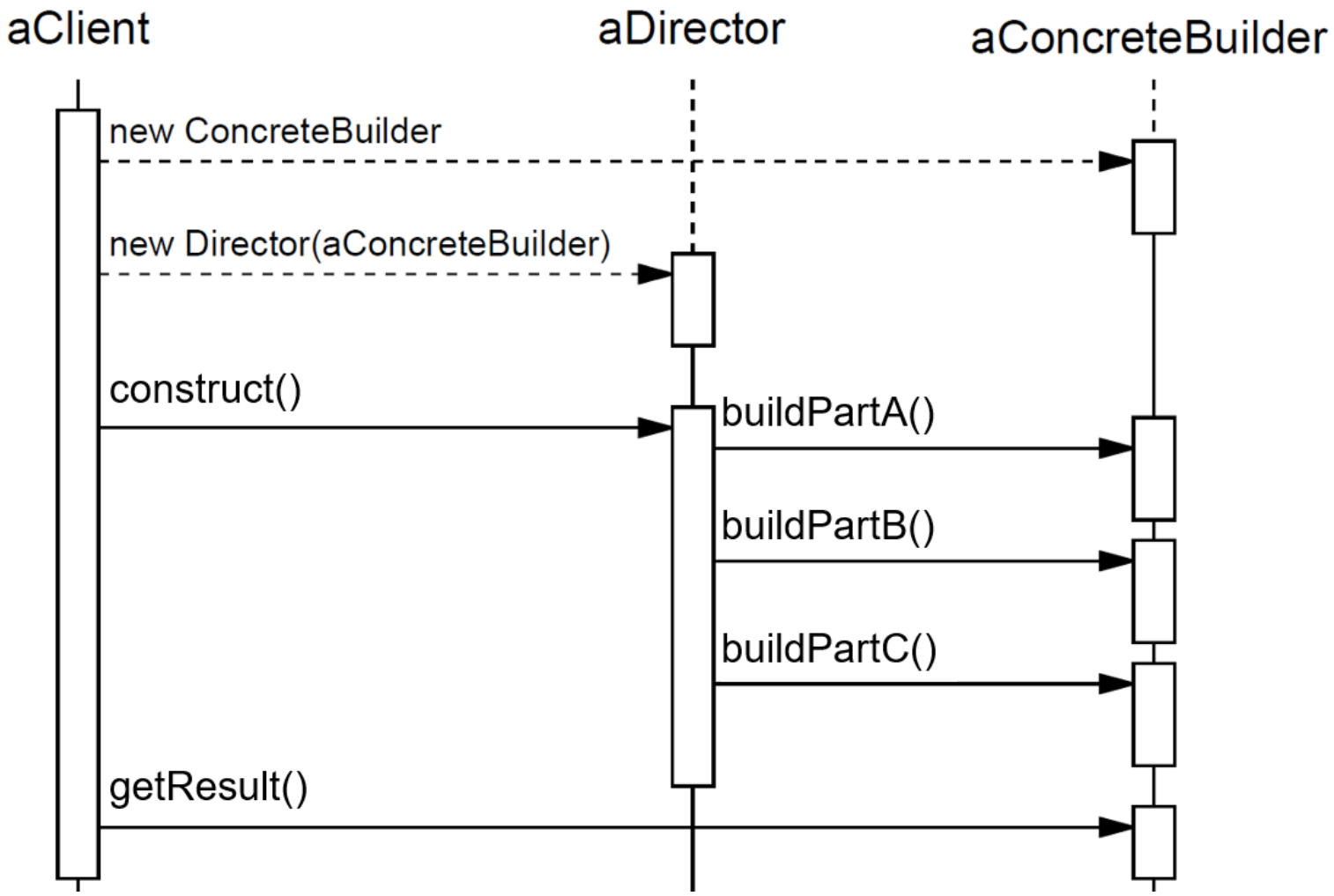
Structure and participants



Structure and participants



Collaborations



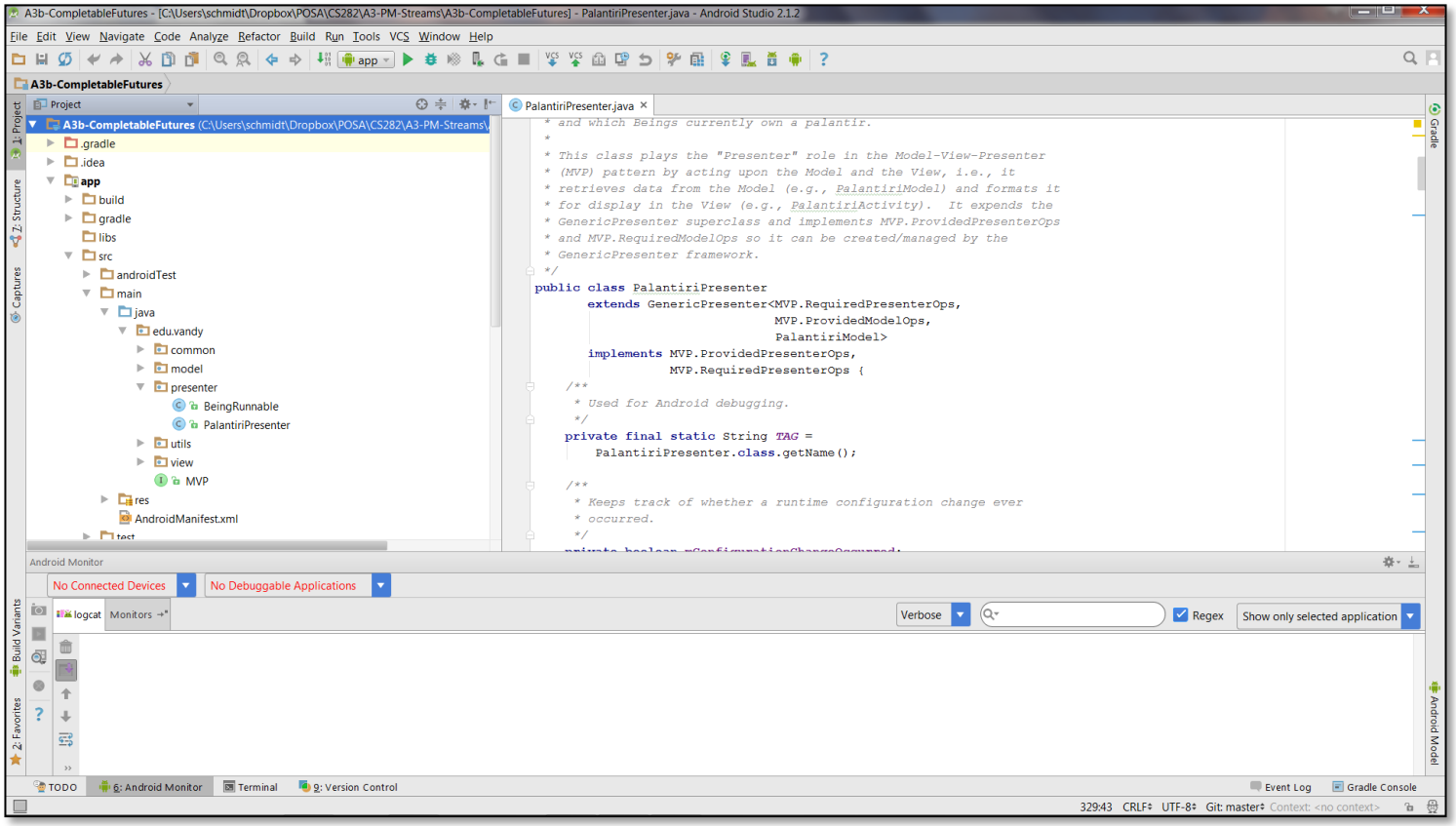
The Builder Pattern

Implementation in Java

Douglas C. Schmidt

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.
- Know how to implement the *Builder* pattern in Java.

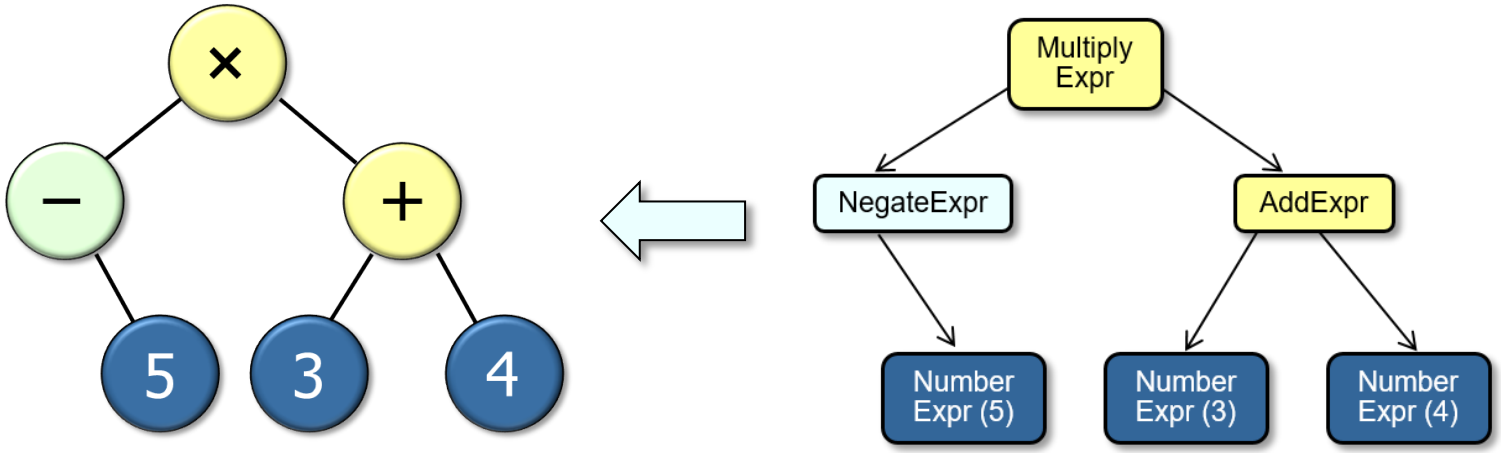


Douglas C. Schmidt

Implementing the Builder Pattern in Java

Builder example in Java

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

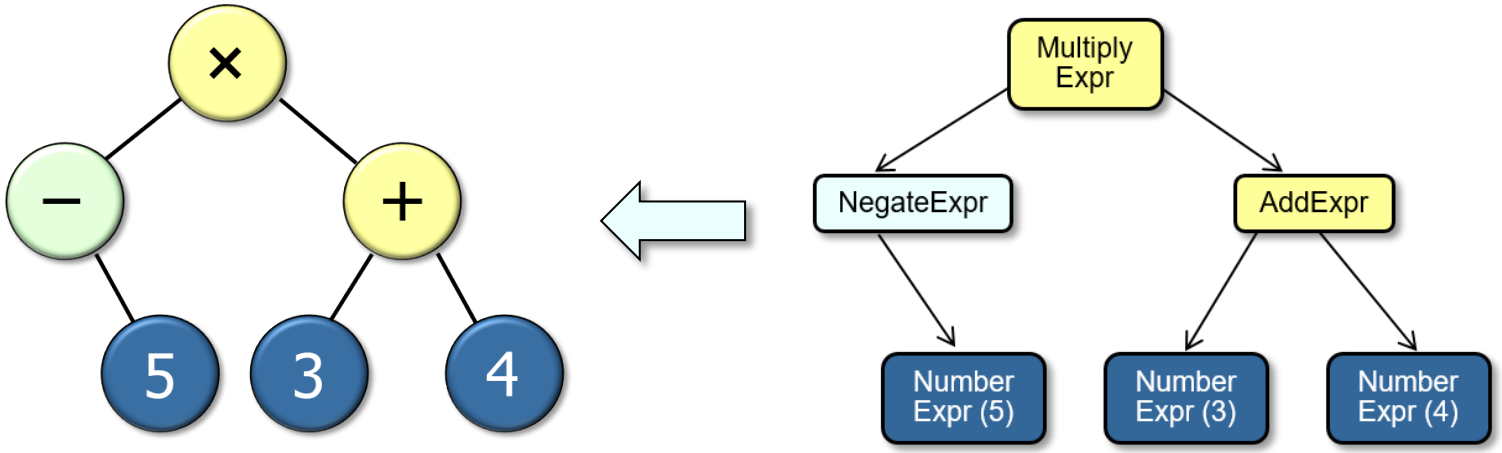



```
class PostOrderInterpreter extends InterpreterImpl {
  protected ExpressionTree buildExpressionTree(Expr parseTree) {
    return expressionTreeFactory
                        .makeExpressionTree(parseTree.build());
  }
```

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

Builder example in Java

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

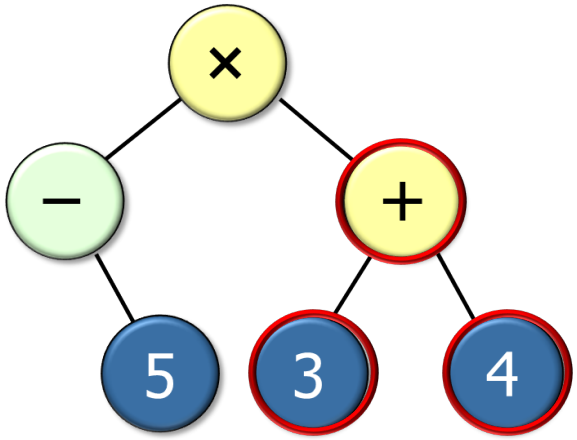


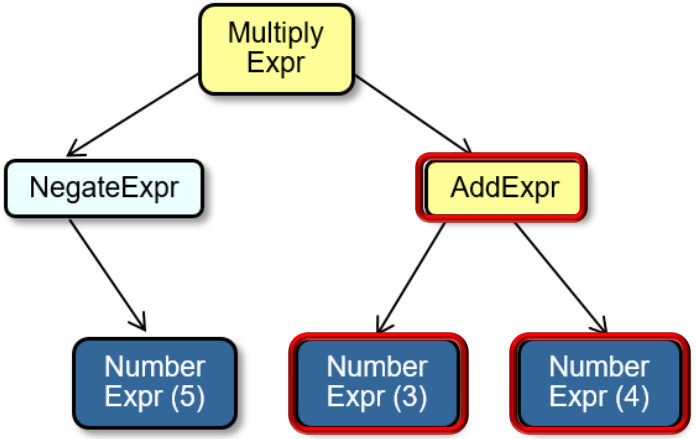

```
class PostOrderInterpreter extends InterpreterImpl {
  protected ExpressionTree buildExpressionTree(Expr parseTree) {
    return expressionTreeFactory
                        .makeExpressionTree(parseTree.build());
  }
```

Invoke a recursive expression tree build, starting with a root expr in the parse tree created by `PostOrderInterpreter`

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);
  }
  ...
```

The Builder Pattern

Other Considerations

Douglas C. Schmidt

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.
- Know how to implement the *Builder* pattern in Java.
- Be aware of other considerations when applying the *Builder* pattern.

Douglas C. Schmidt

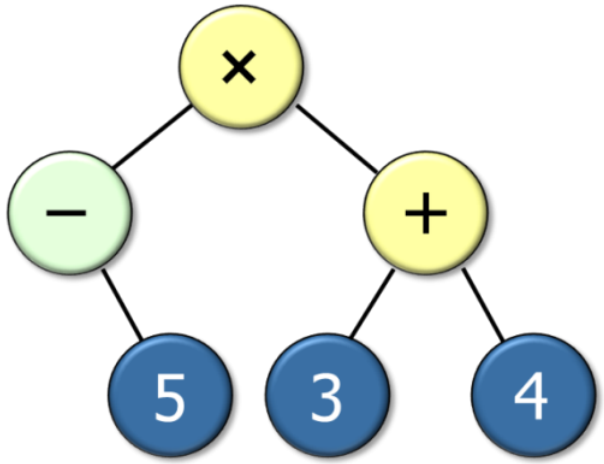
Other Considerations of the Builder Pattern

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());
  ...
}
```



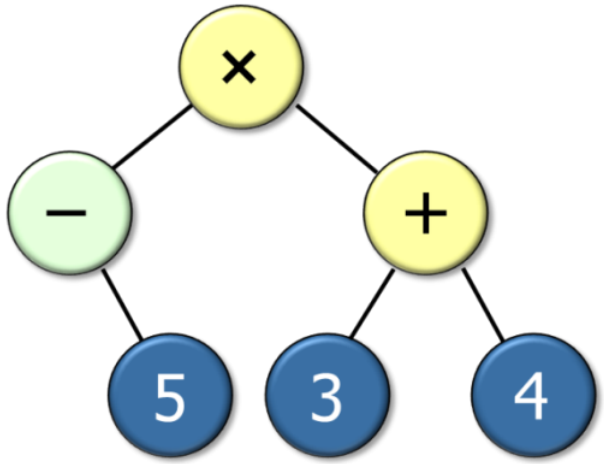
Consequences

+ Finer control over the construction process

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

Every composite node controls how it's constructed.

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

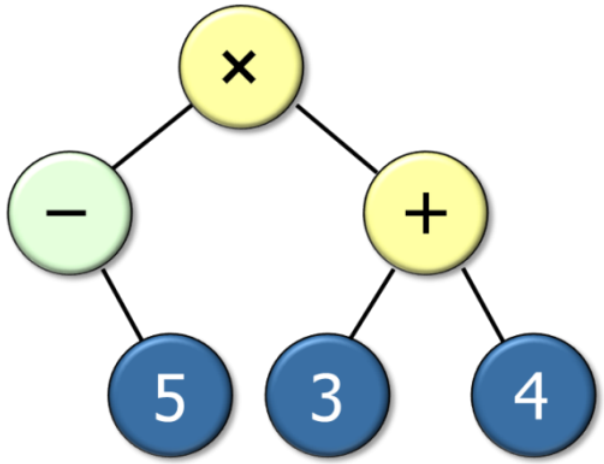


Consequences

+ Can vary a product's internal representation

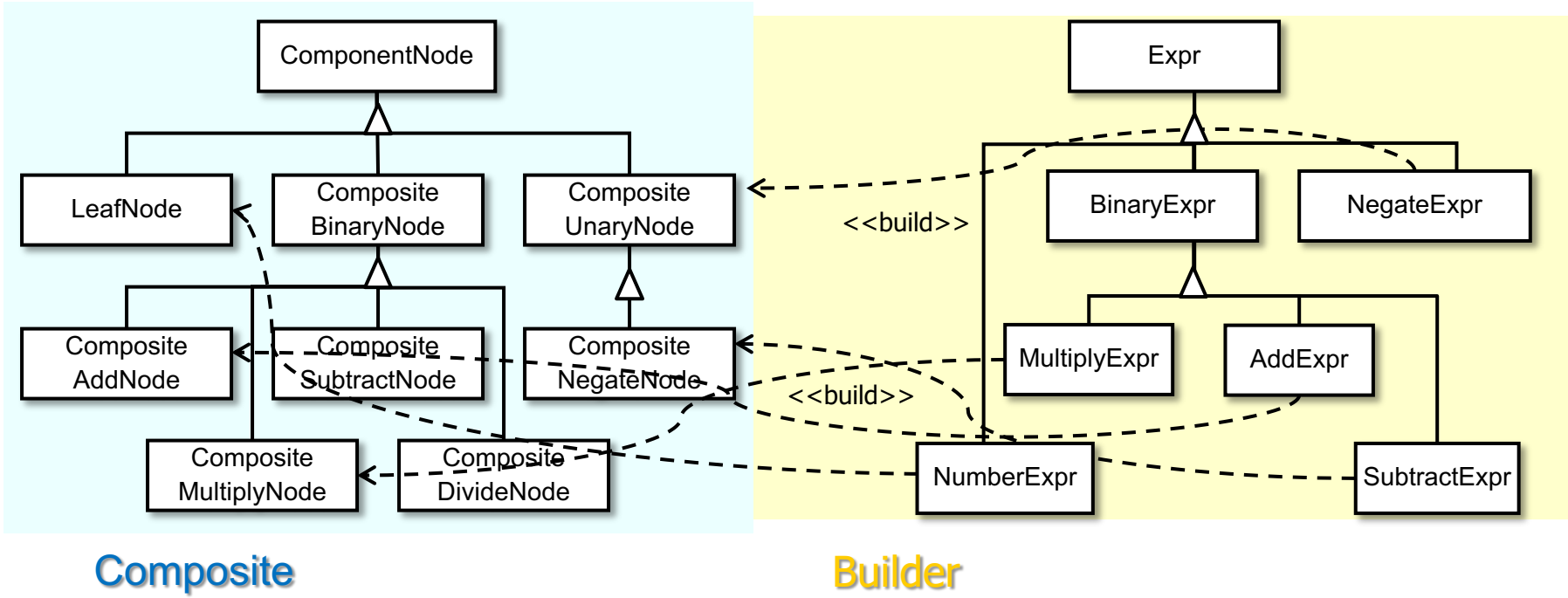
```
class NumberExpr extends Expr {
  ComponentNode build() {
    return new TreeNode(item);
  }
  ...
}
```

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



Consequences

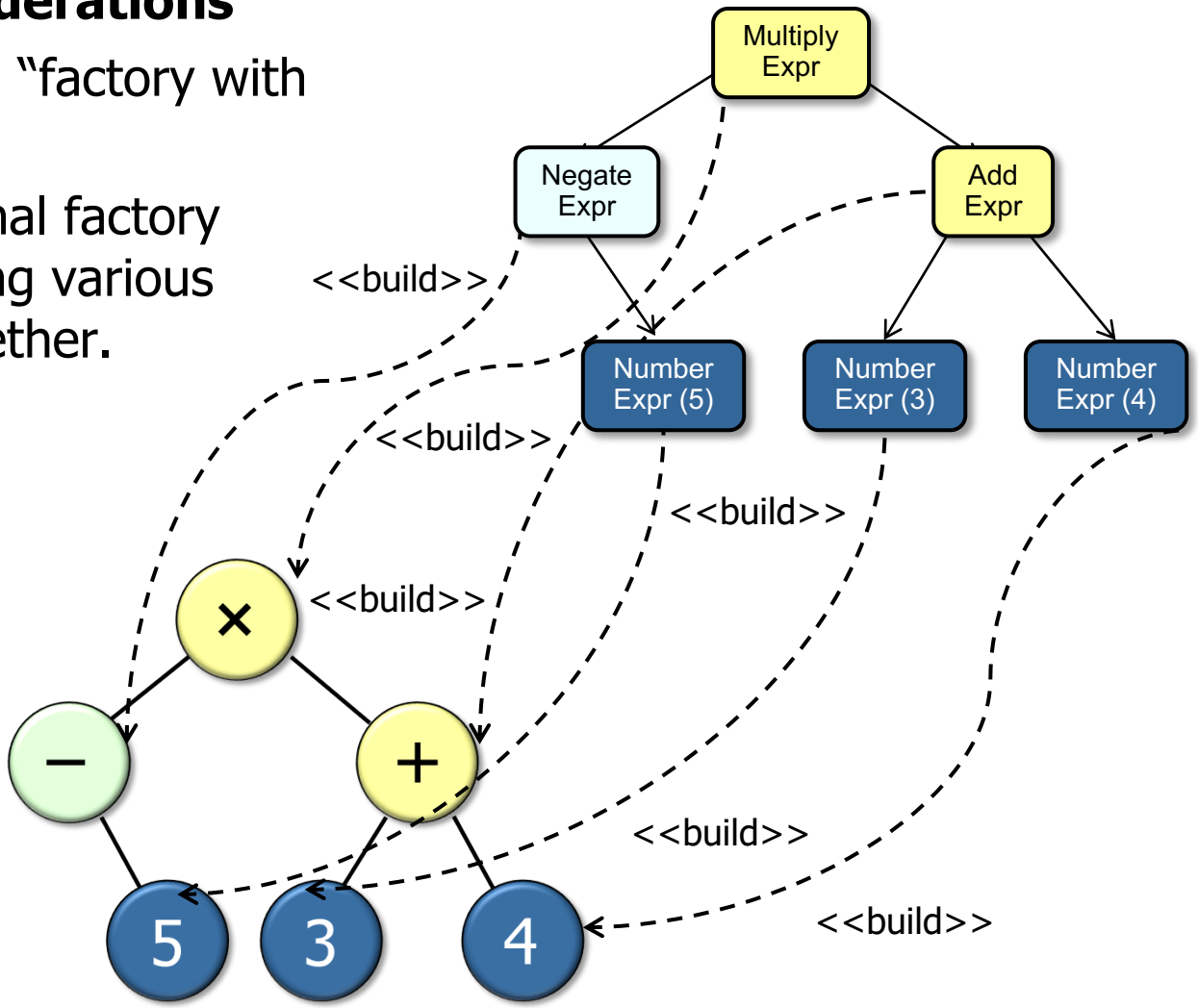
– May involve a lot of classes and class interdependencies



Knowledge of patterns helps to reduce the "surface area" of these many classes.

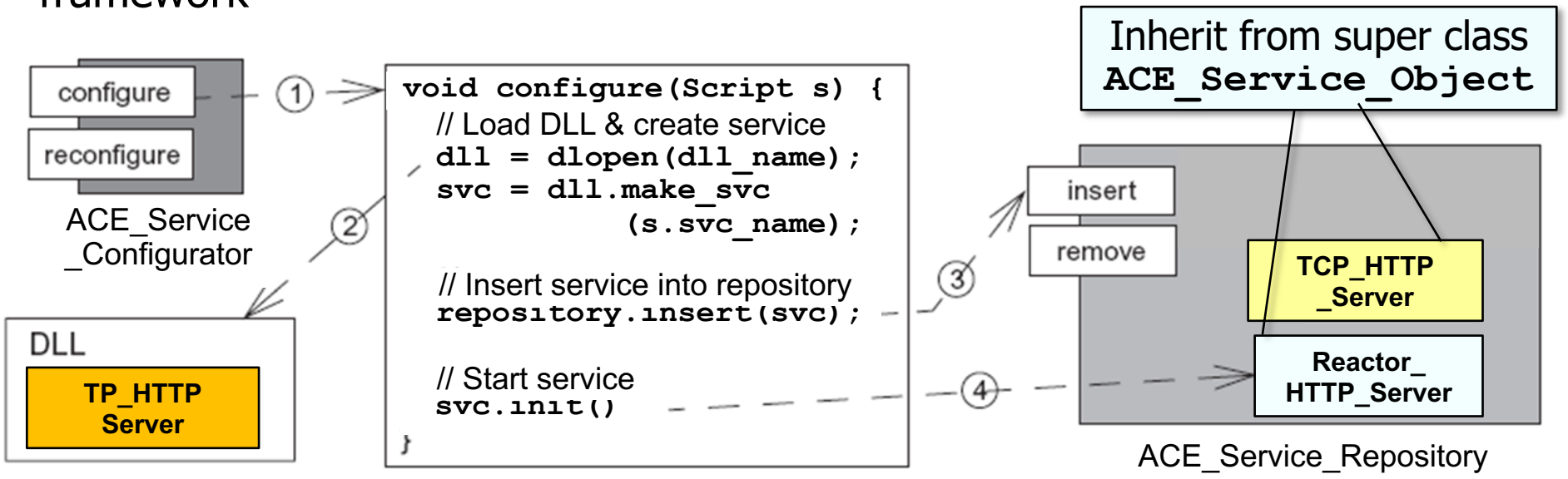
Implementation considerations

- 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



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

Known uses

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

```
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); }
  ...
```

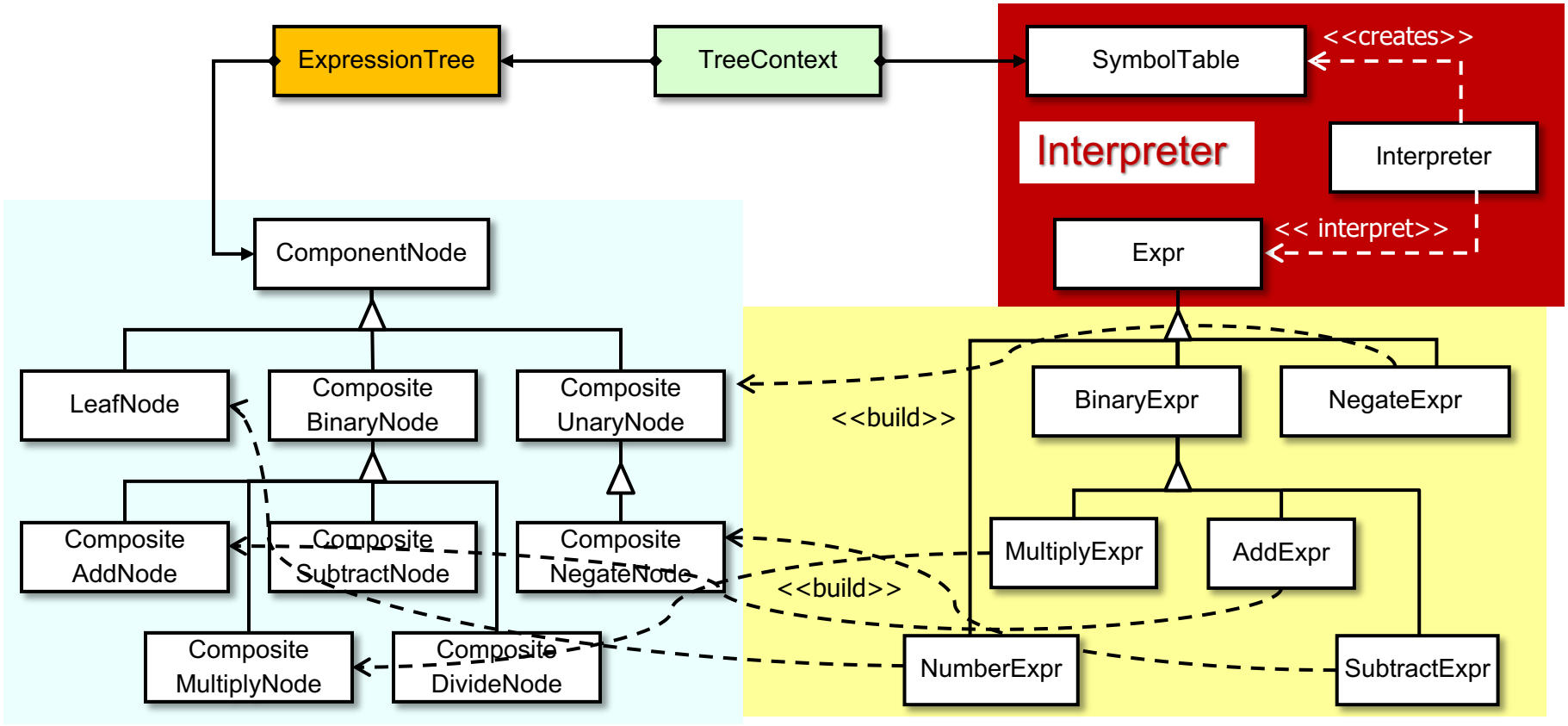
```
public class Test {
  public static void
  main(String[] a) {
    Builder task =
      new Builder()
      .setDesc("Builder test").setSummary("Cool!").build();
    ...
  } ...
```

Note the use of "fluent interface" API design style.

See en.wikipedia.org/wiki/Fluent_interface

Summary of the Builder Pattern

- *Builder* recursively builds the *Composite*-based expression tree data structure by processing the *Interpreter*-based parse tree generated from user input.



Interpreter, Builder, and Composite are a useful "pattern sequence."

See www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf

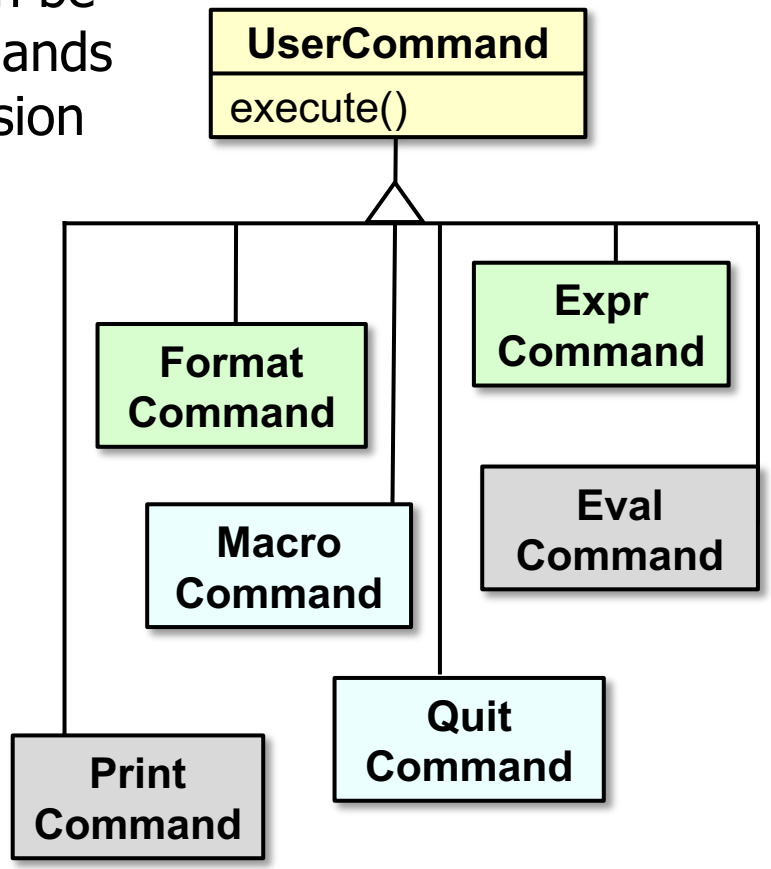
The Command Pattern

Motivating Example

Douglas C. Schmidt

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.

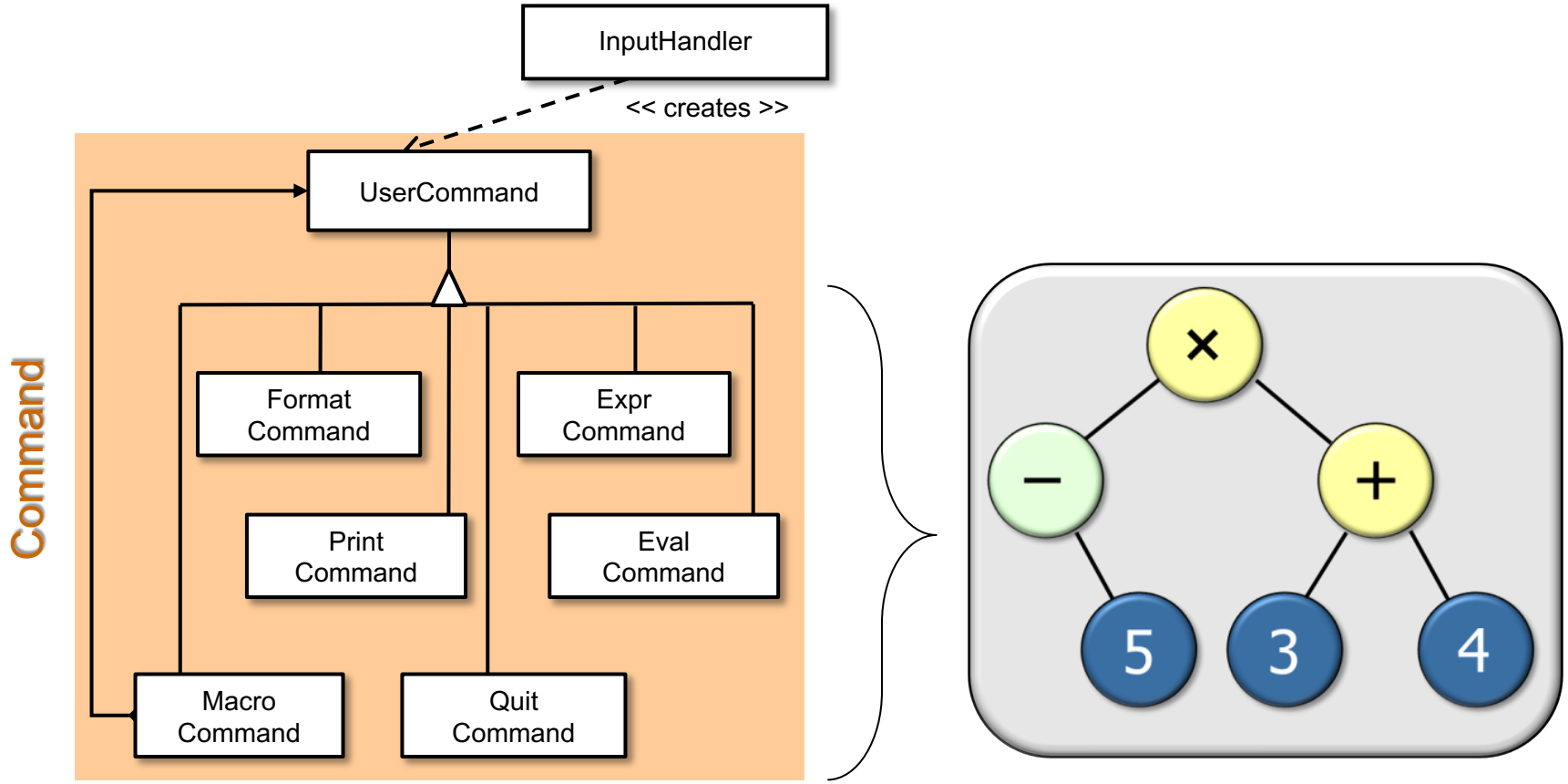


Douglas C. Schmidt

Motivating the Need for the Command Pattern in the Expression Tree App

A Pattern for Objectifying User Requests

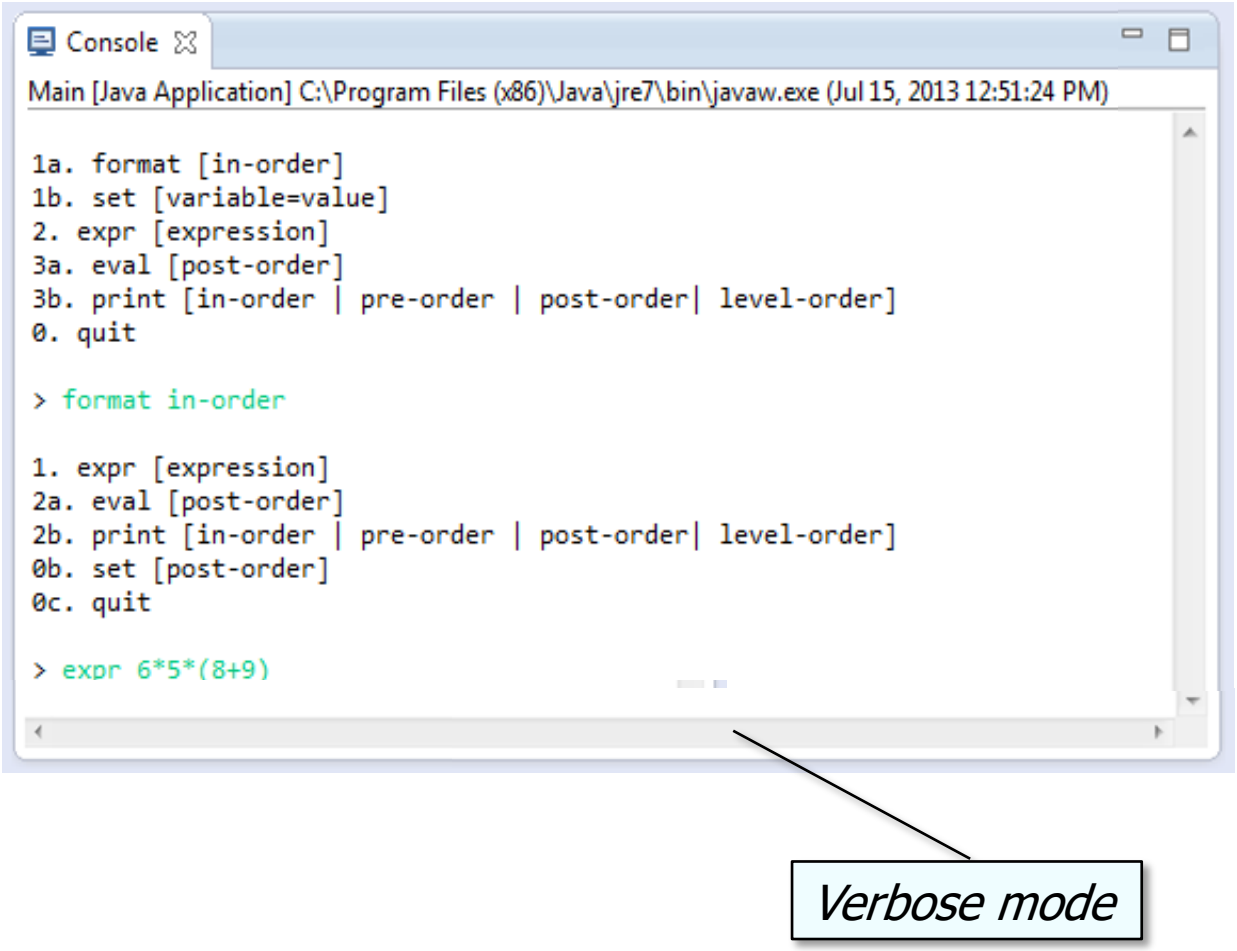
***Purpose**: Define objectified actions that enable users to perform command requests consistently and extensibly in the expression tree processing app.*



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

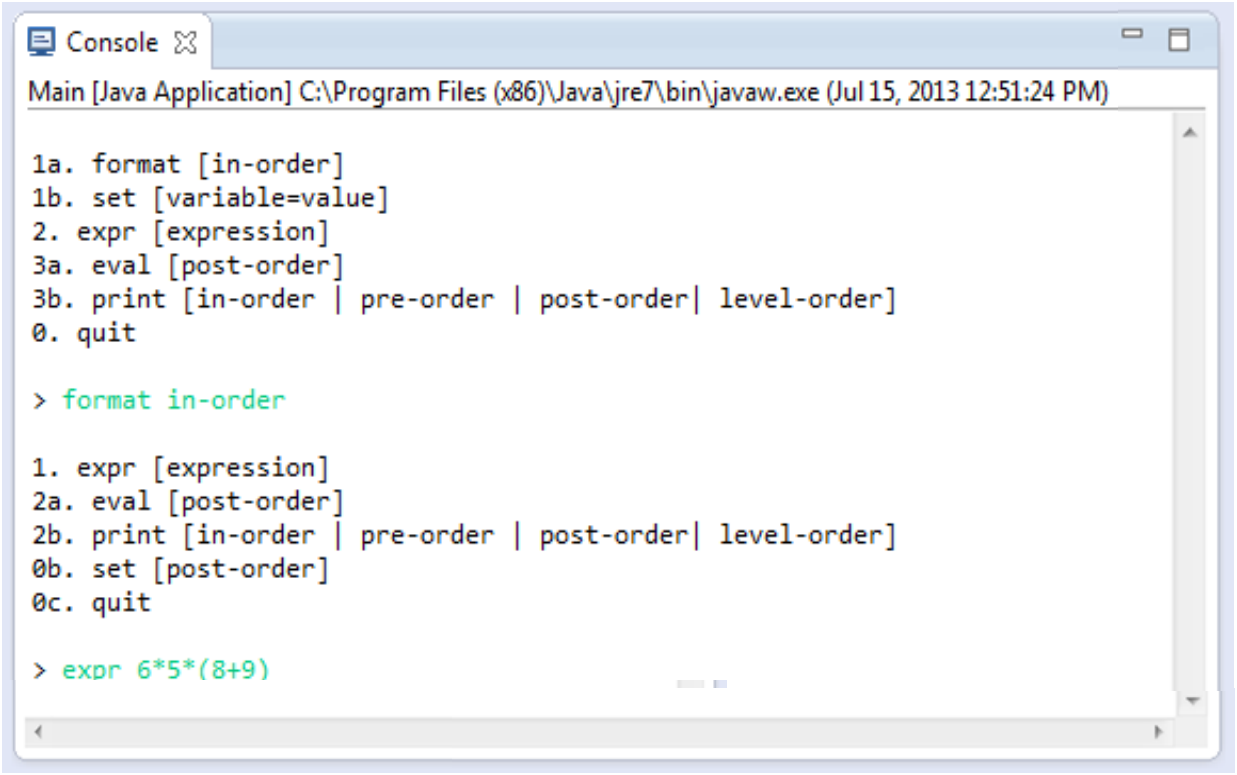
Context: OO Expression Tree Processing App

- Verbose mode supports user command execution

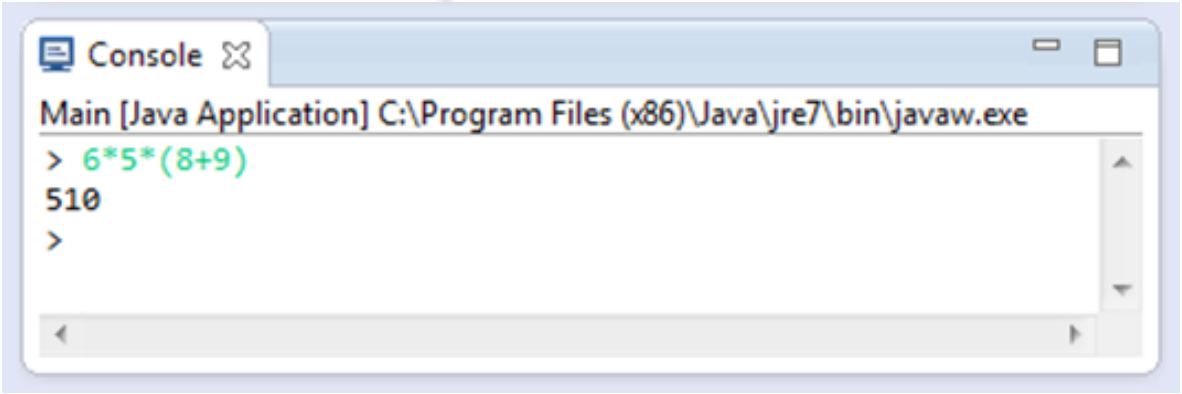


Context: OO Expression Tree Processing App

- Succinct mode supports macro commands

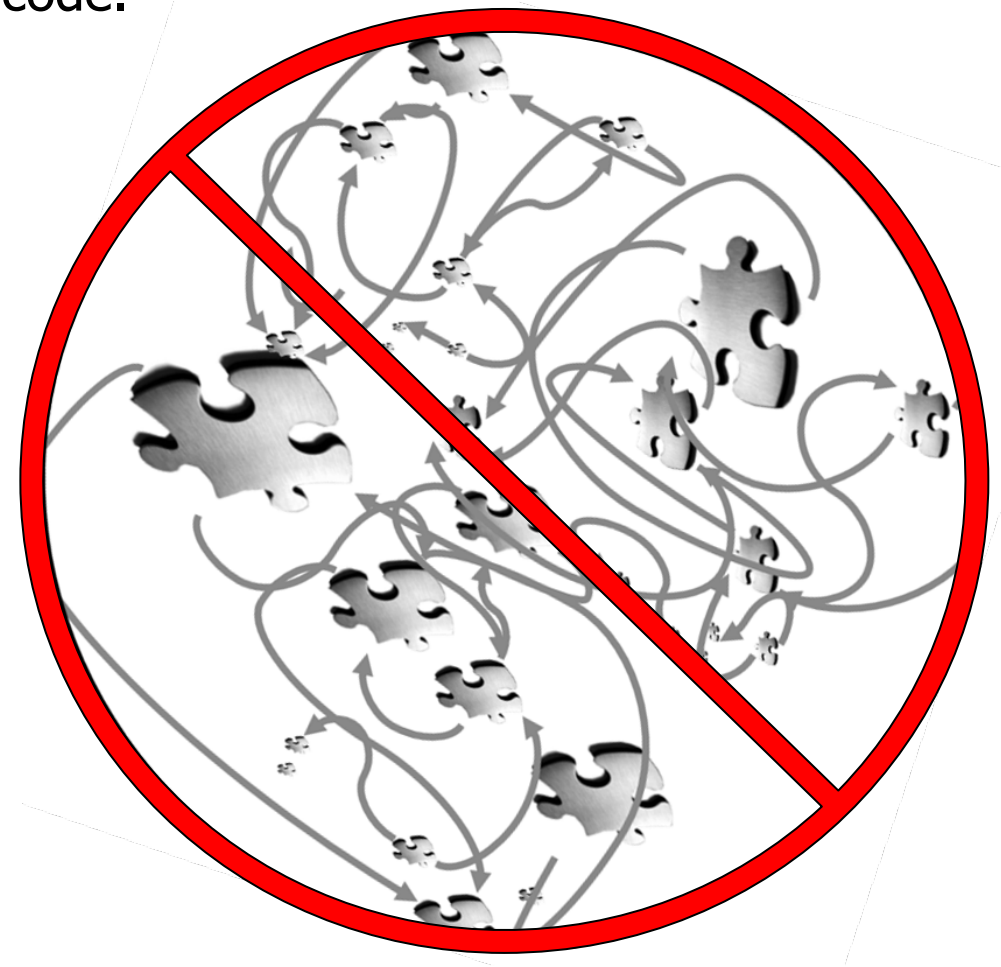


Succinct mode



Problem: Scattered/Fixed User Request Implementations

- It's hard to maintain implementations of user-requested commands that are scattered throughout the source code.

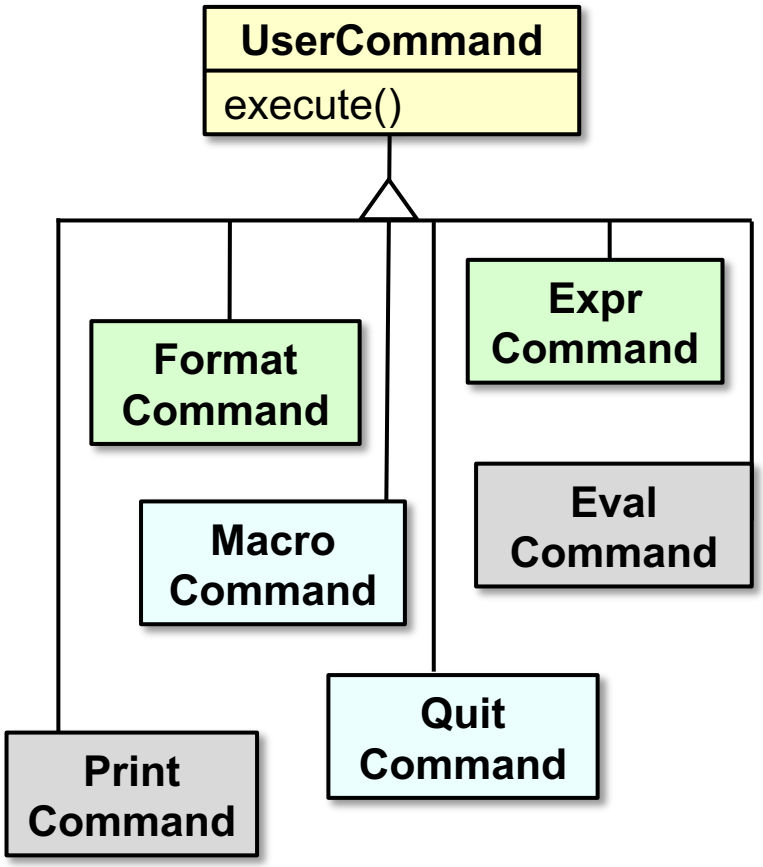


Problem: Scattered/Fixed User Request Implementations

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

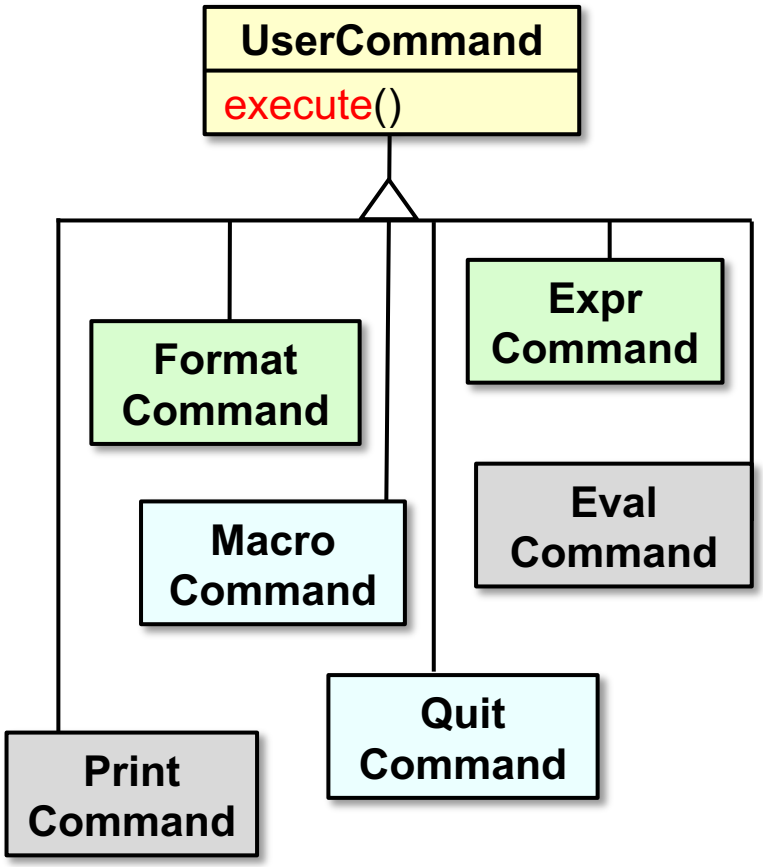
Solution: Encapsulate User Requests as Commands

- Create a hierarchy of **`UserCommand`** subclasses



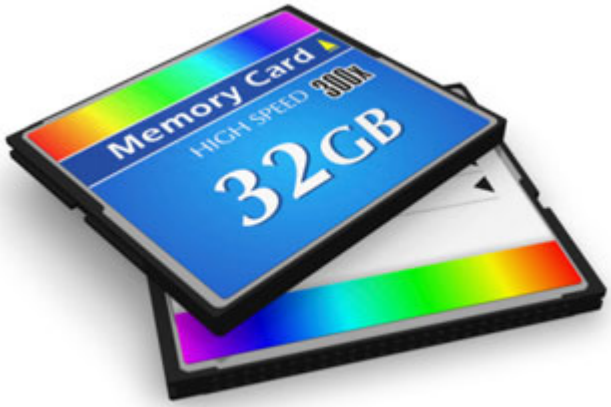
Solution: Encapsulate User Requests as Commands

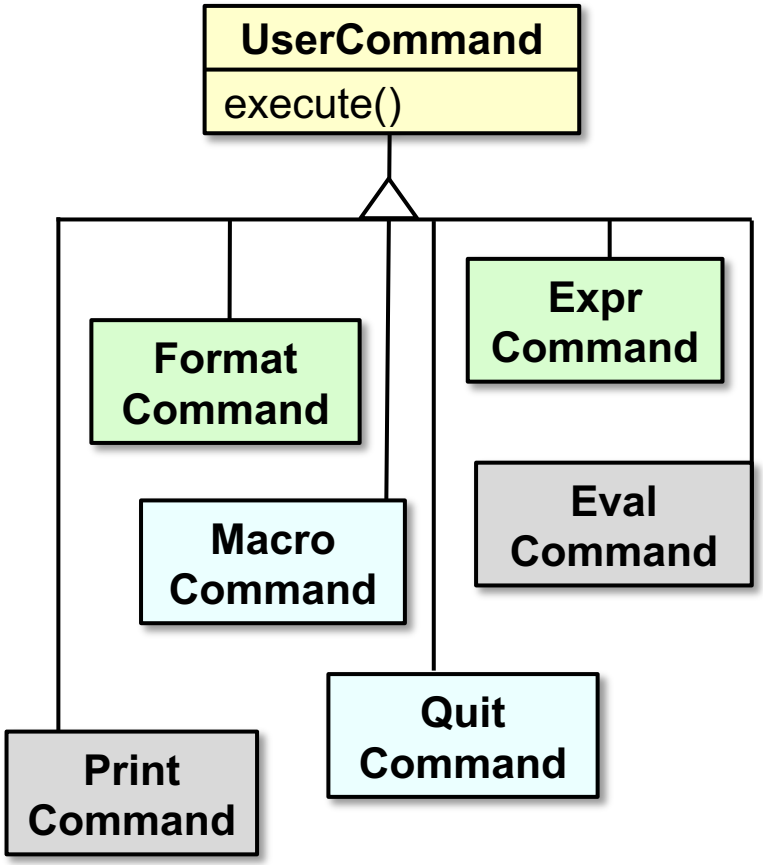
- Create a hierarchy of **`UserCommand`** subclasses, each containing:
  - A command method (**`execute()`**)



Solution: Encapsulate User Requests as Commands

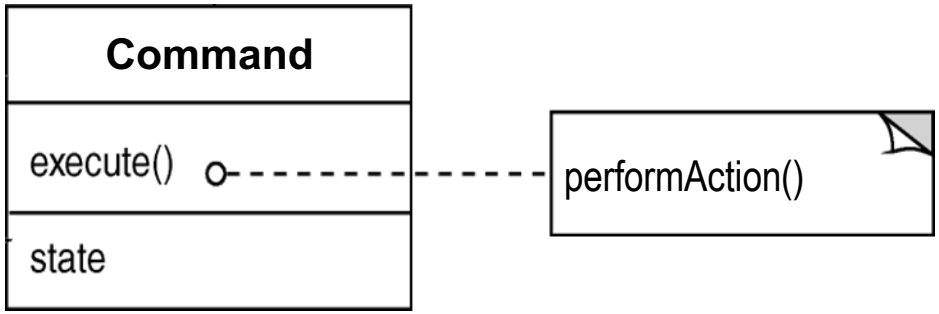
- Create a hierarchy of `UserCommand` subclasses, each containing:
  - A command method (`execute()`)
  - The state needed by the command





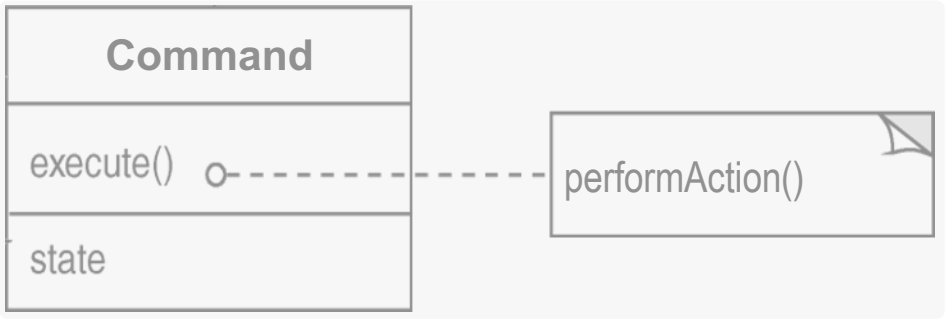
Solution: Encapsulate User Requests as Commands

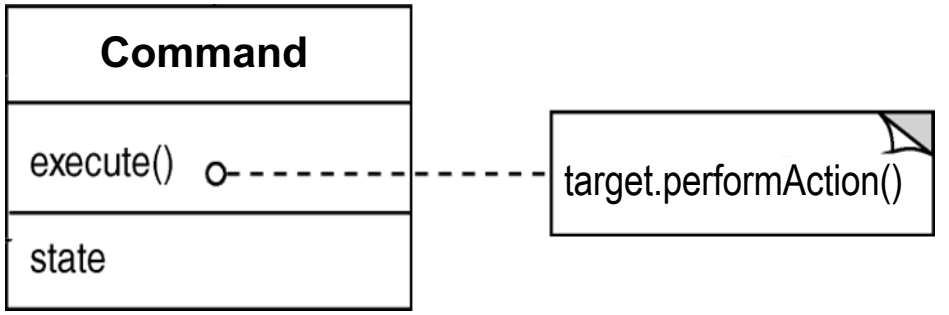
- A Command object may:
  - Implement the command itself



Solution: Encapsulate User Requests as Commands

- A Command object may:
  - Implement the operation itself
  - Or forward the command's implementation to other object(s)





UserCommand Class Overview

- Defines an abstract super class that performs a user-requested command on an expression tree when it's executed

Class methods

```
void execute()
void printValidCommands()
```

UserCommand Class Overview

- Defines an abstract super class that performs a user-requested command on an expression tree when it's executed

Class methods

These methods are defined by subclasses

```
void execute()
void printValidCommands()
```

UserCommand Class Overview

- Defines an abstract super class that performs a user-requested command on an expression tree when it's executed

Class methods

```
void execute()
void printValidCommands()
```

- **Commonality**: provides a common API for expression tree commands
- **Variability**: subclasses of `UserCommand` can vary depending on the commands requested by user input

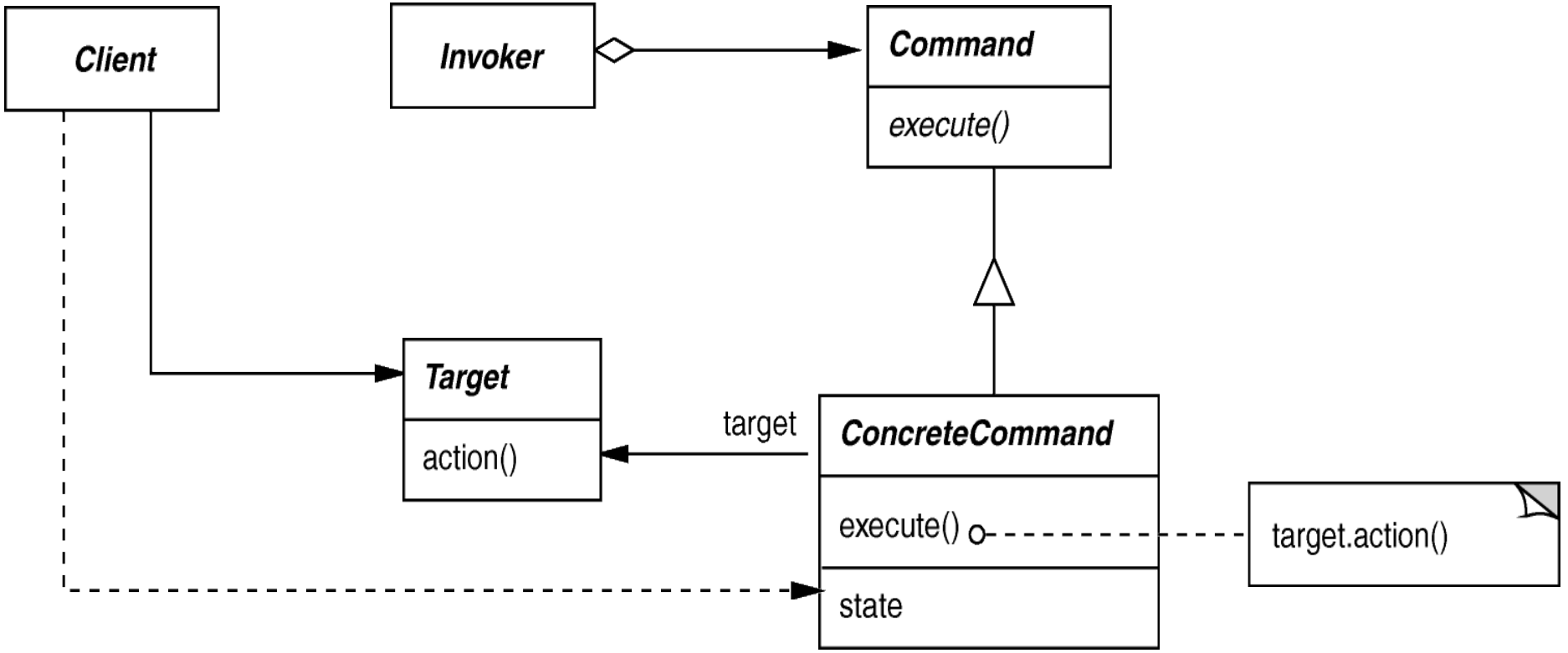
The Command Pattern

Structure and Functionality

Douglas C. Schmidt

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.

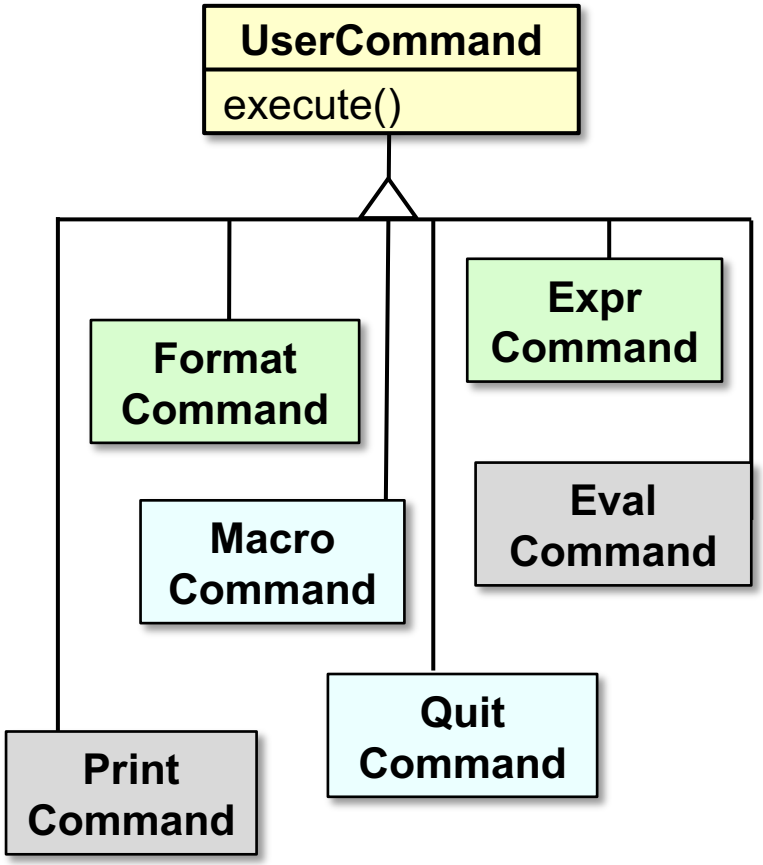


Douglas C. Schmidt

Structure and Functionality of the Command Pattern

Intent

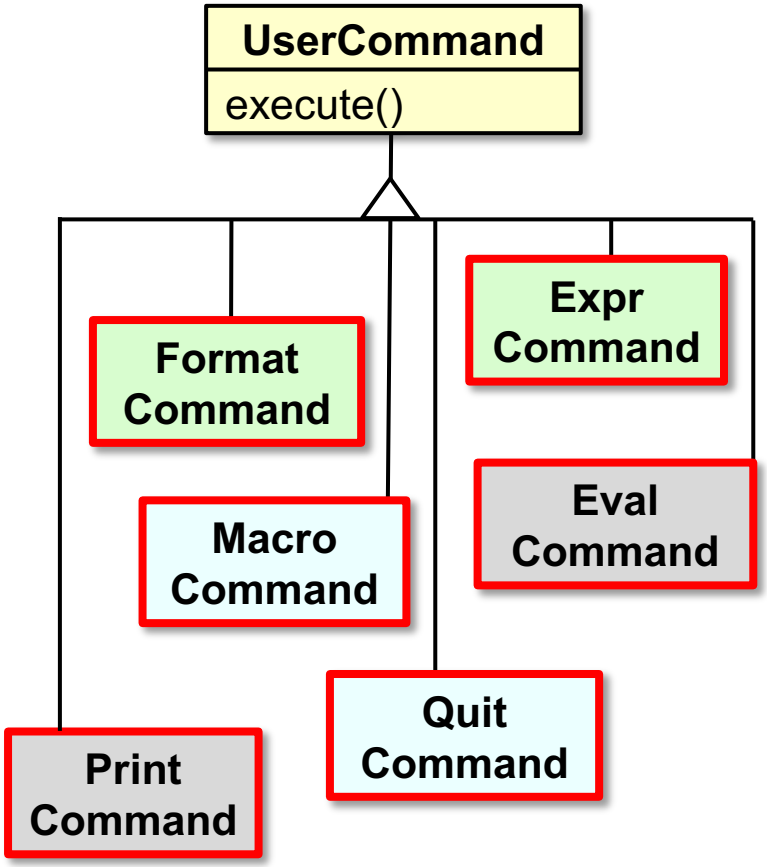
- Encapsulate the request for a service as an object



See en.wikipedia.org/wiki/Command_pattern

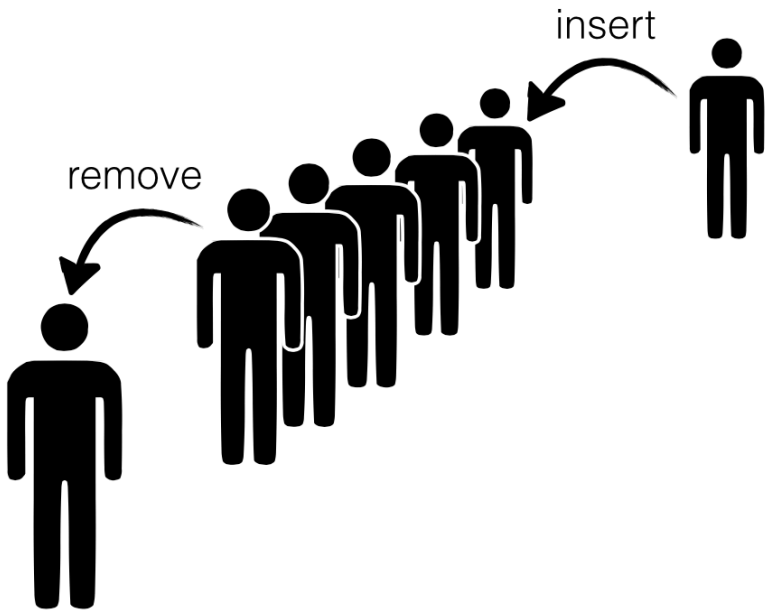
Applicability

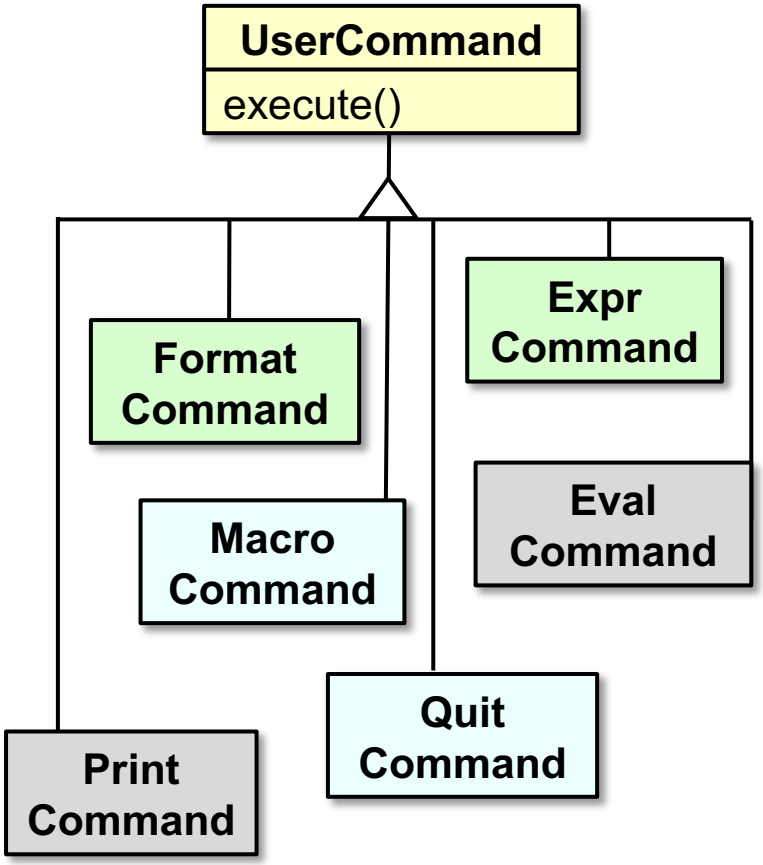
- Want to parameterize objects with an action to perform



Applicability

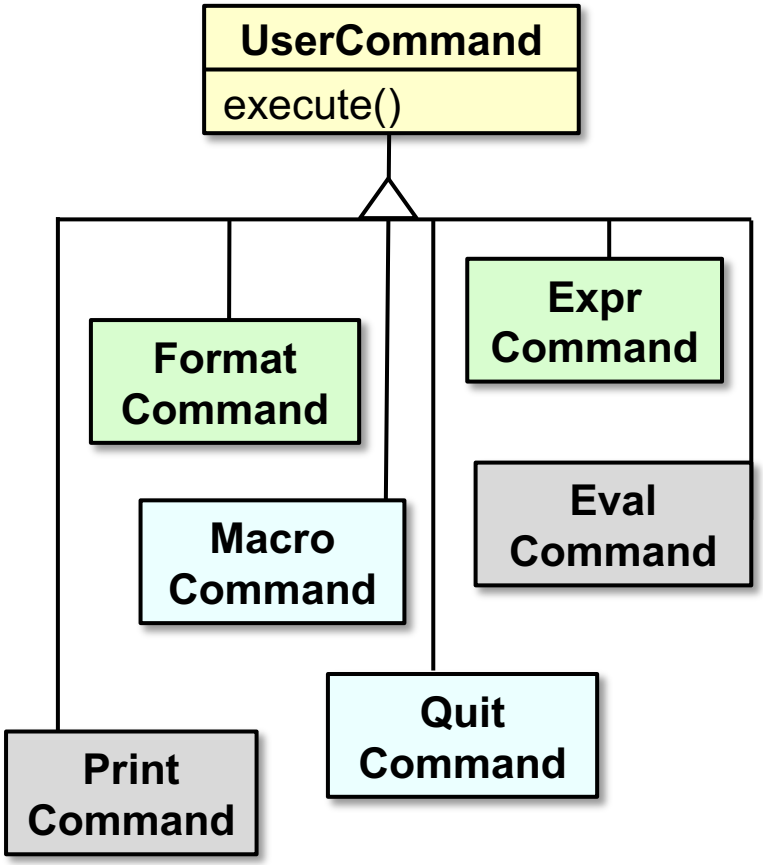
- Want to parameterize objects with an action to perform
- Want to specify, queue, and execute requests at different times



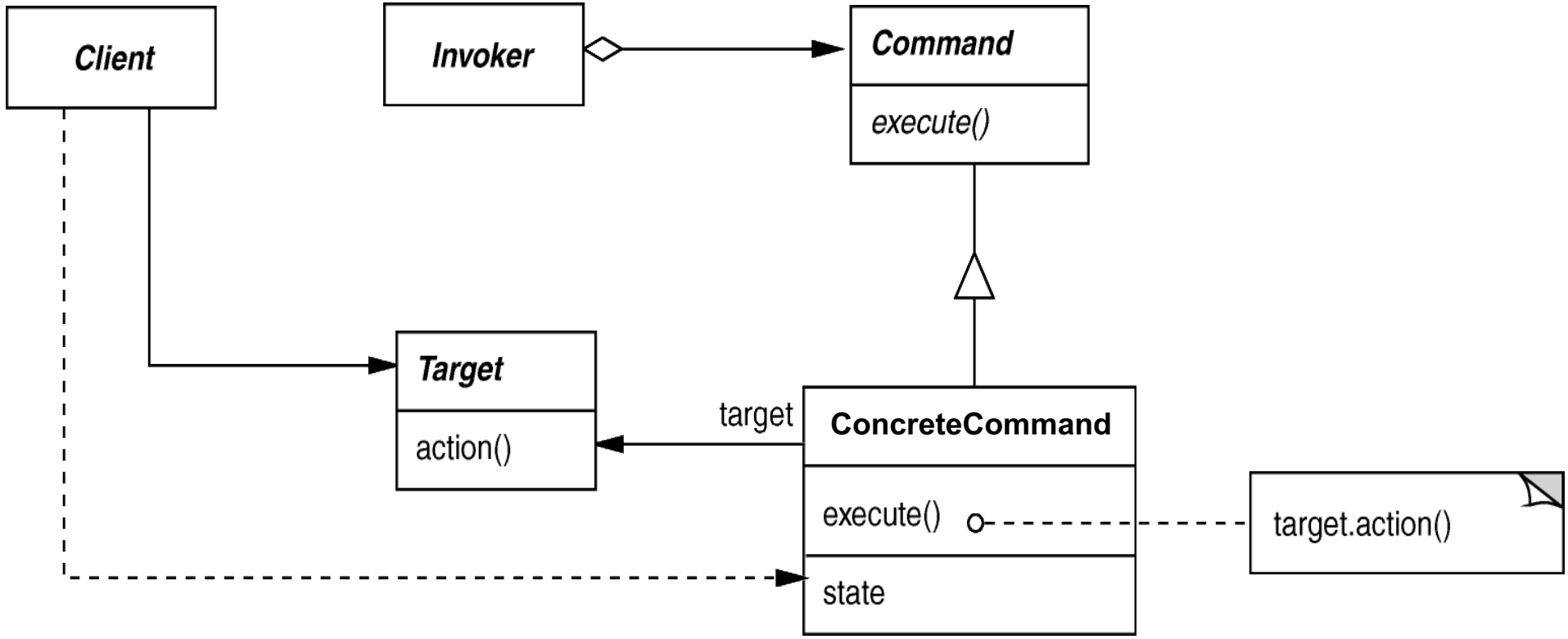


Applicability

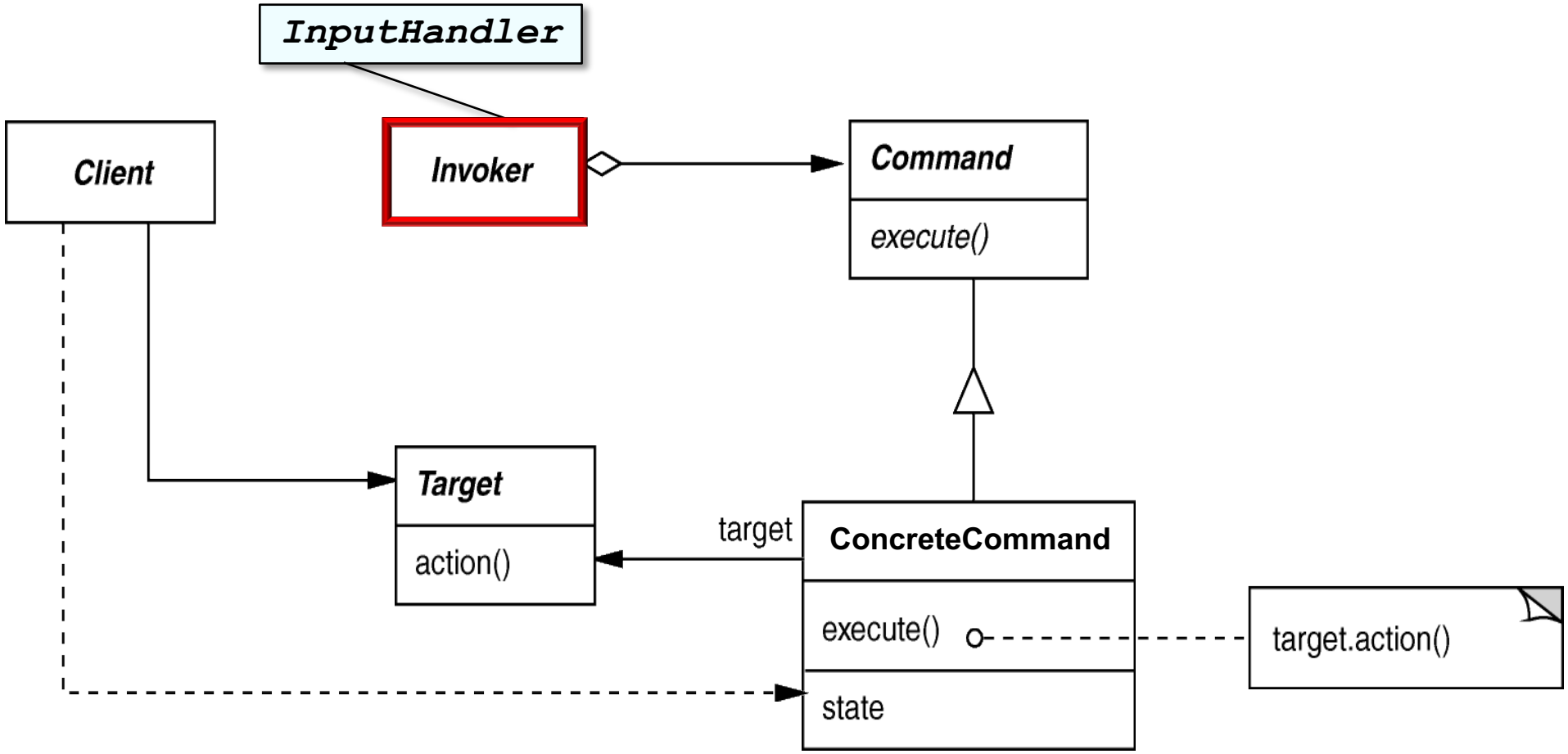
- Want to parameterize objects with an action to perform
- Want to specify, queue, and execute requests at different times
- Want to support multilevel undo/redo



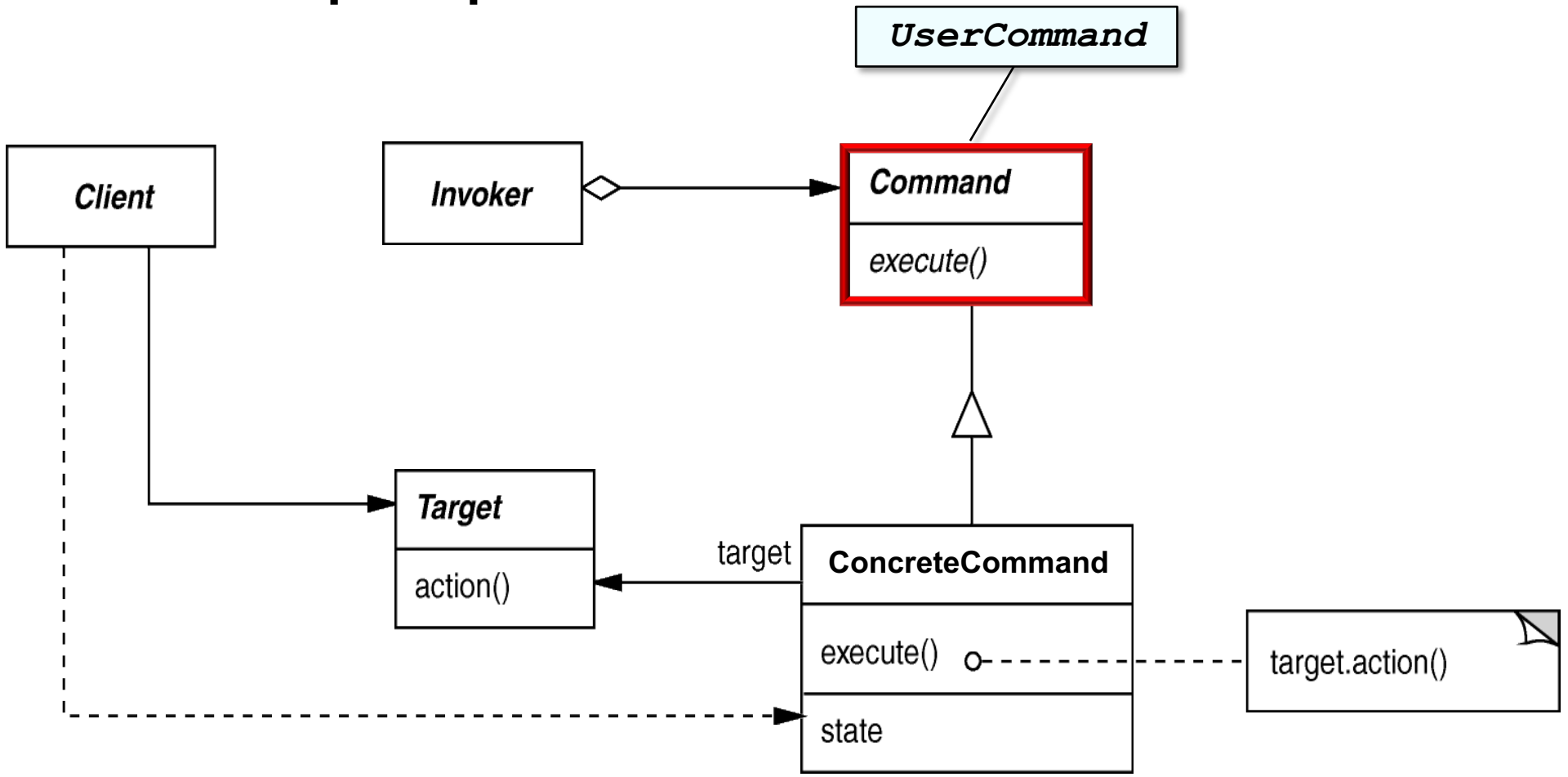
Structure and participants



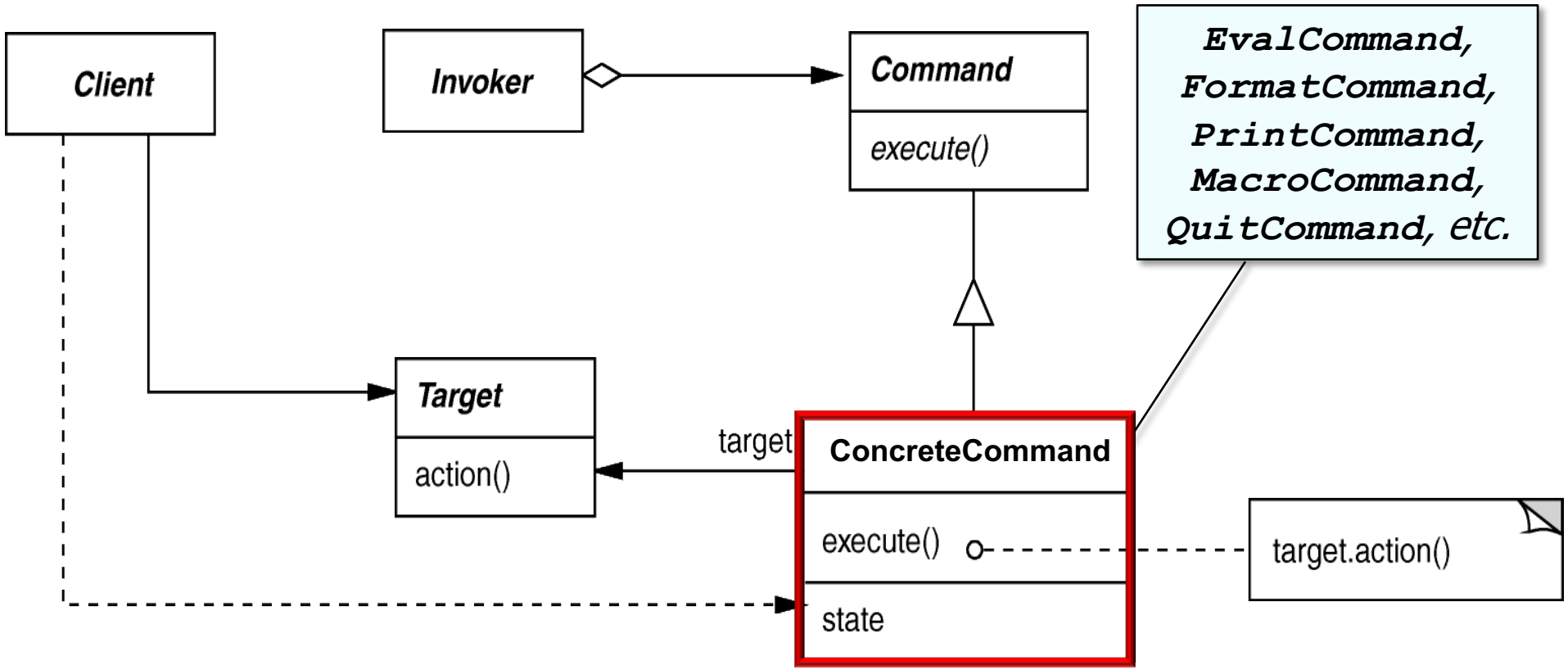
Structure and participants



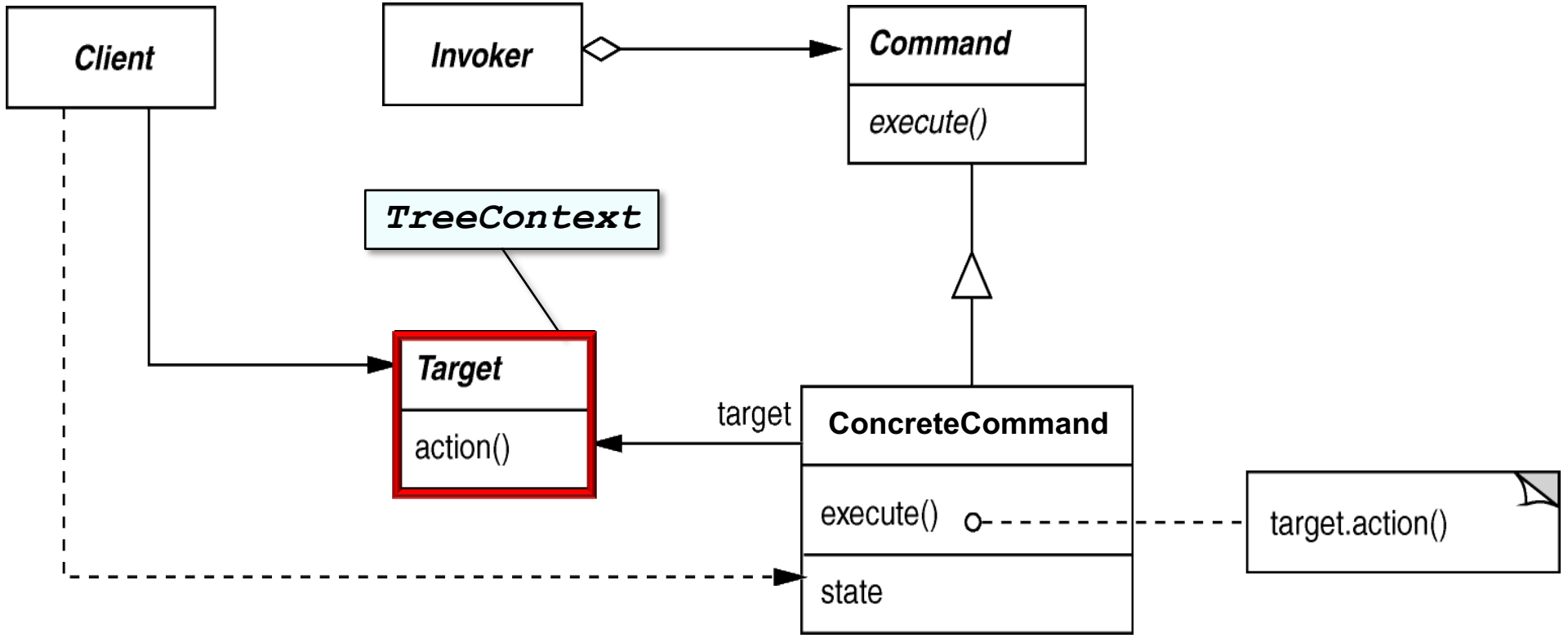
Structure and participants



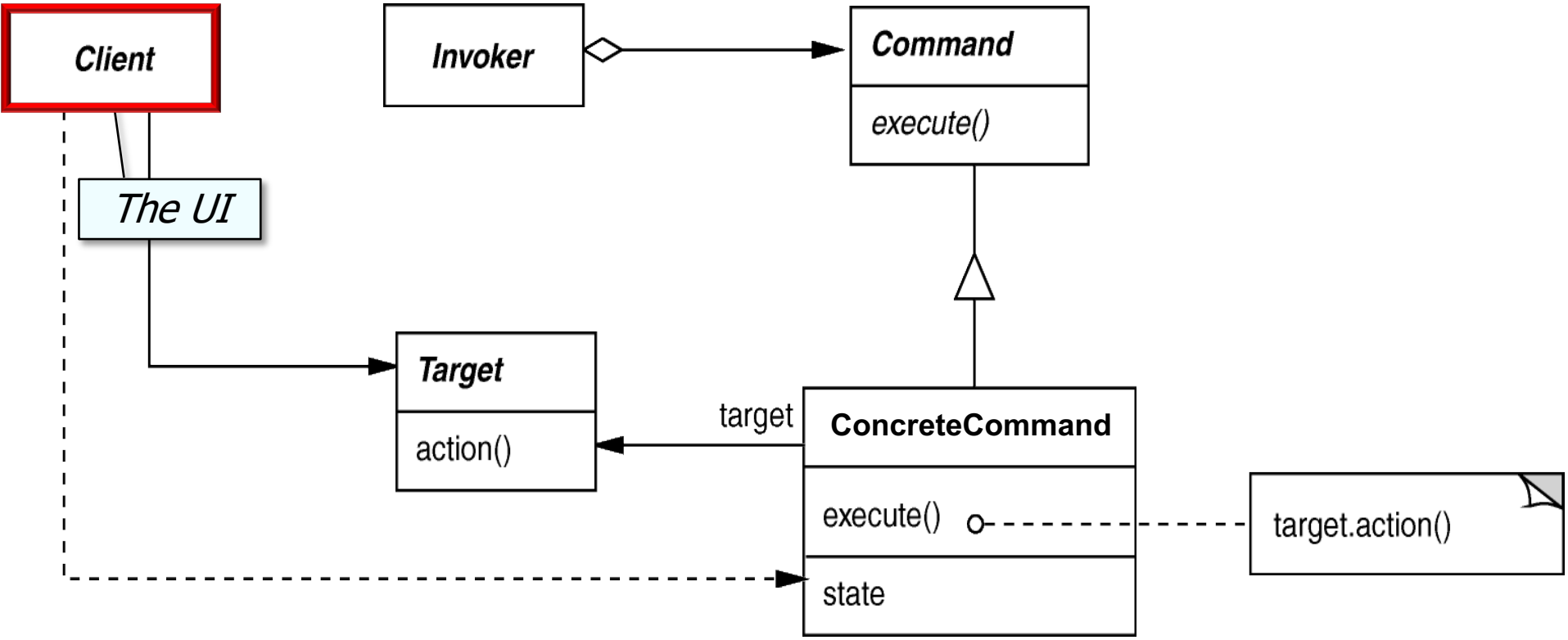
Structure and participants



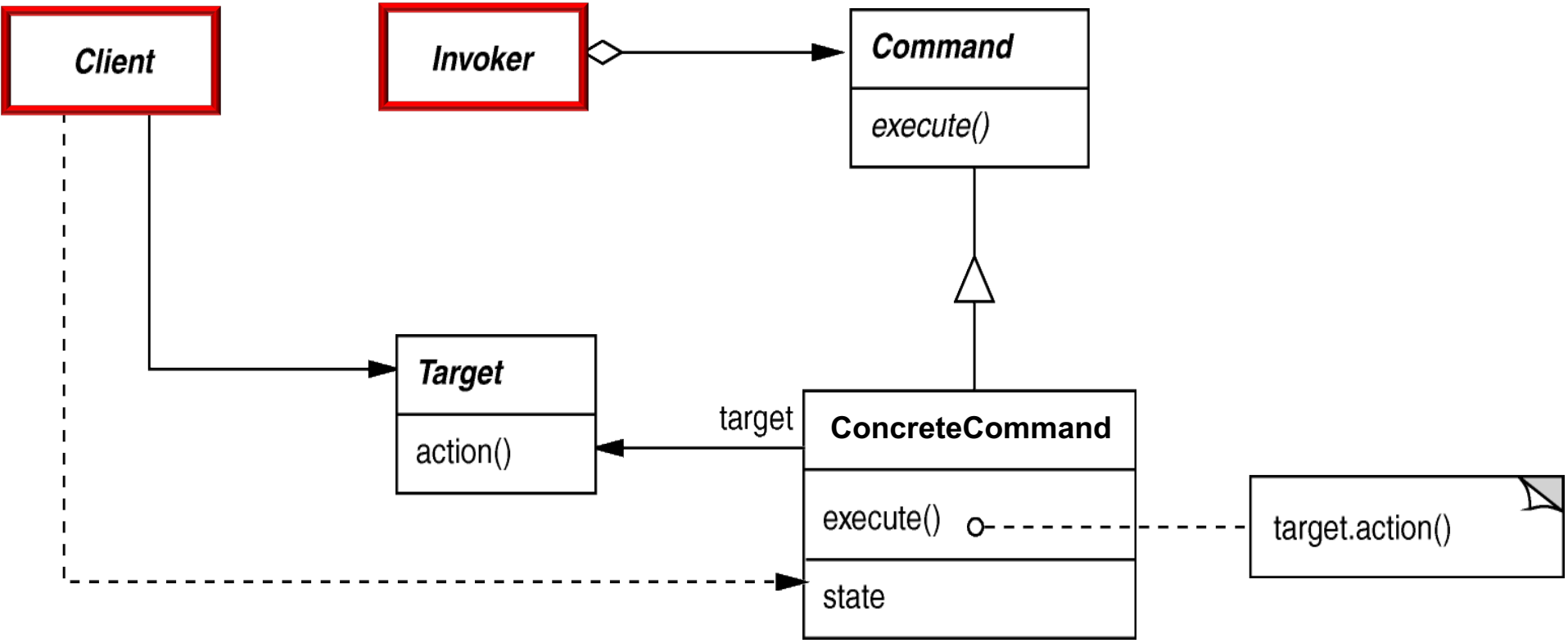
Structure and participants



Structure and participants



Structure and participants



The **`Client`** and **`Invoker`** objects may be the same or different.

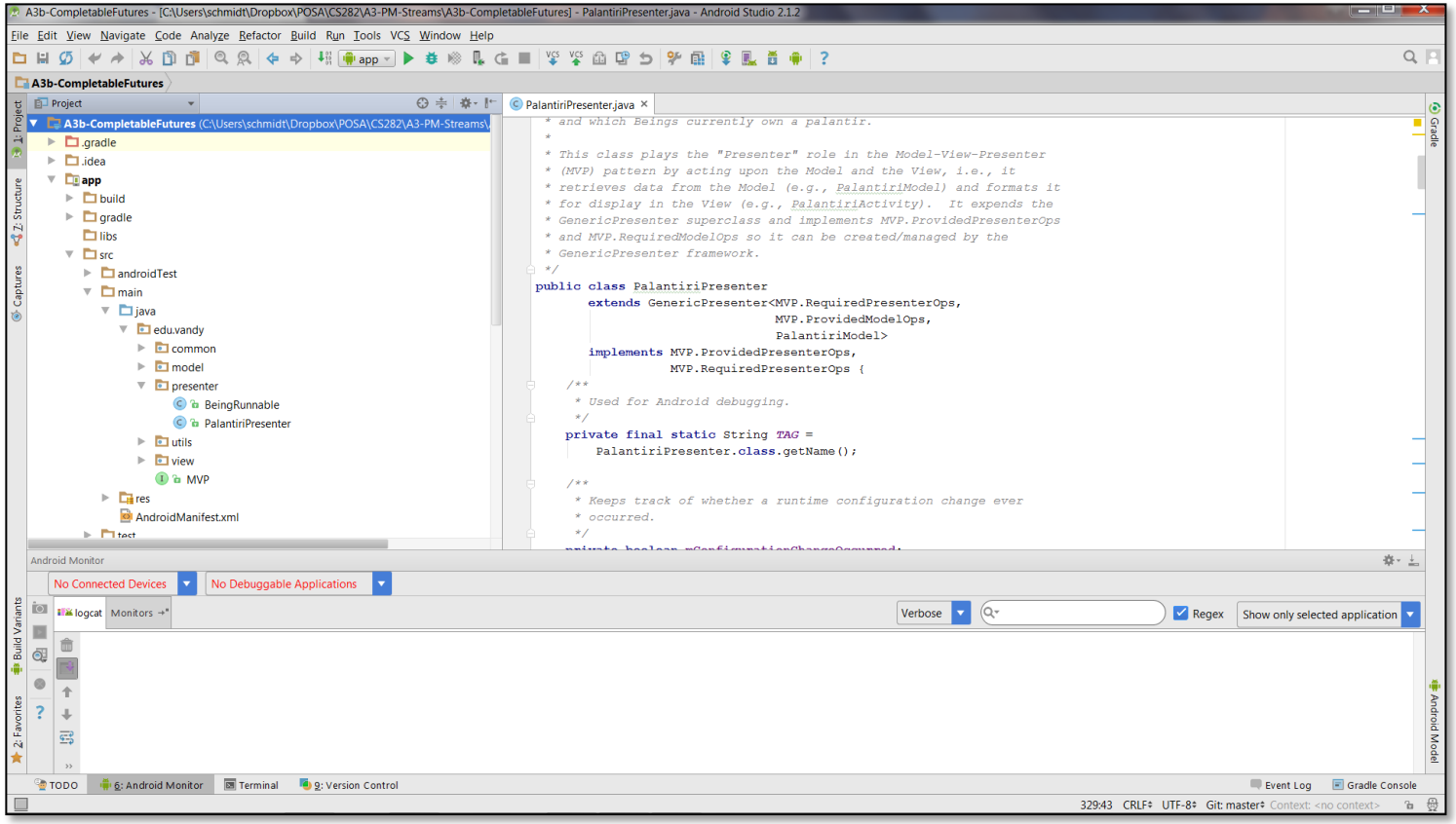
The Command Pattern

Implementation in Java

Douglas C. Schmidt

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.
- Know how to implement the *Command* pattern in Java.



Douglas C. Schmidt

Implementing the Command Pattern in Java

Command example in Java

- Encapsulate the execution of a command object that sets the desired input expression.
  - e.g., "−5×(3+4)"

```
public class ExprCommand
       extends UserCommand {
  private String mExpr;


  ExprCommand(TreeContext context,
              String newexpr) {
       super(context);
       mExpr = newexpr;
  }


  public void execute() {
     mTreeContext.expr(mExpr);
  }
```

See ExpressionTree/CommandLine/src/expressiontree/commands

Command example in Java

- Encapsulate the execution of a command object that sets the desired input expression.
  - e.g., "−5×(3+4)"

```
public class ExprCommand
       extends UserCommand {
  private String mExpr;
```

Store the requested expression

```
  ExprCommand(TreeContext context,
              String newexpr) {
       super(context);
       mExpr = newexpr;
  }

  public void execute() {
     mTreeContext.expr(mExpr);
  }
```

Command example in Java

- Encapsulate the execution of a command object that sets the desired input expression.
  - e.g., "−5×(3+4)"

```
public class ExprCommand
       extends UserCommand {
  private String mExpr;


  ExprCommand(TreeContext context,
              String newexpr) {
       super(context);
       mExpr = newexpr;
  }

  public void execute() {
     mTreeContext.expr(mExpr);
  }
```

Provide appropriate TreeContext and requested expression

Command example in Java

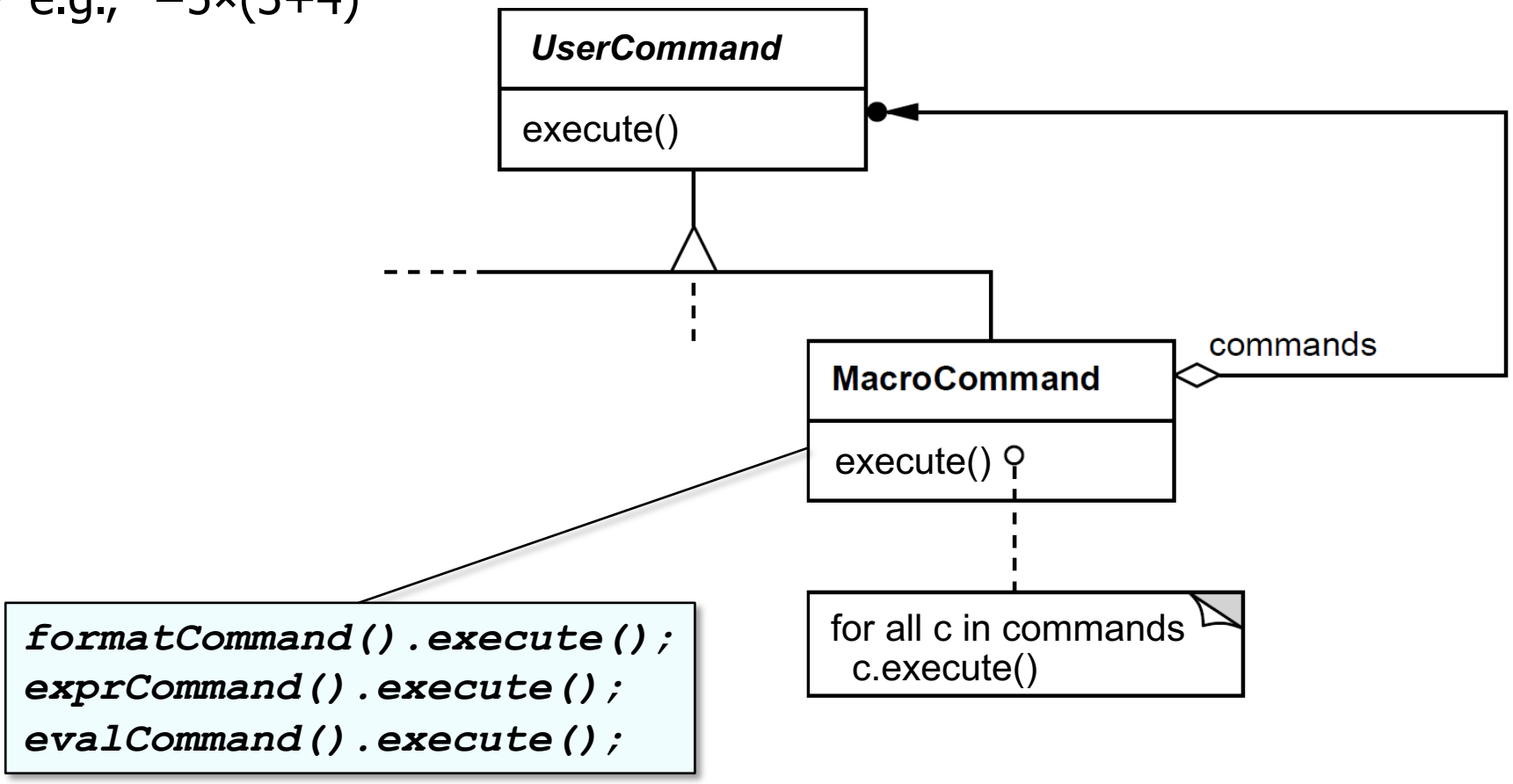
- Encapsulate the execution of a command object that sets the desired input expression.
  - e.g., "−5×(3+4)"

```
public class ExprCommand
       extends UserCommand {
  private String mExpr;


  ExprCommand(TreeContext context,
              String newexpr) {
       super(context);
       mExpr = newexpr;
  }


  public void execute() {
     mTreeContext.expr(mExpr);
  }
```

Create the desired expression tree

Command example in Java

- Encapsulate the execution of a sequence of commands as an object, which is used to implement the "succinct mode."
  - e.g., "−5×(3+4)"



See ExpressionTree/CommandLine/src/expressiontree/commands

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 {
    for (UserCommand command : mMacroCommands)
      command.execute();
  }
  ...
```

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<>();
```

List of commands to execute as a macro

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

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

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<>();
```

Constructor initializes the field

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

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

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 {
    for (UserCommand command : mMacroCommands)
      command.execute();
  }
  ...
```

Java for-each loop runs a sequence of commands to implement the "succinct mode"

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

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.
- Know how to implement the *Command* pattern in Java.
- Be aware of other considerations when applying the *Command* pattern.

Douglas C. Schmidt

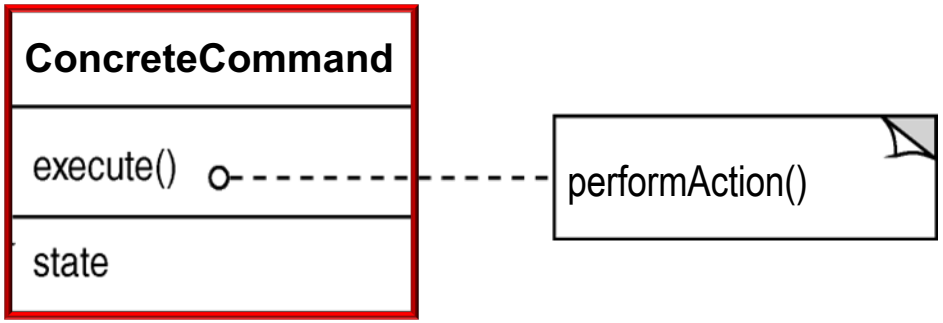
Other Considerations of the Command Pattern

Consequences

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

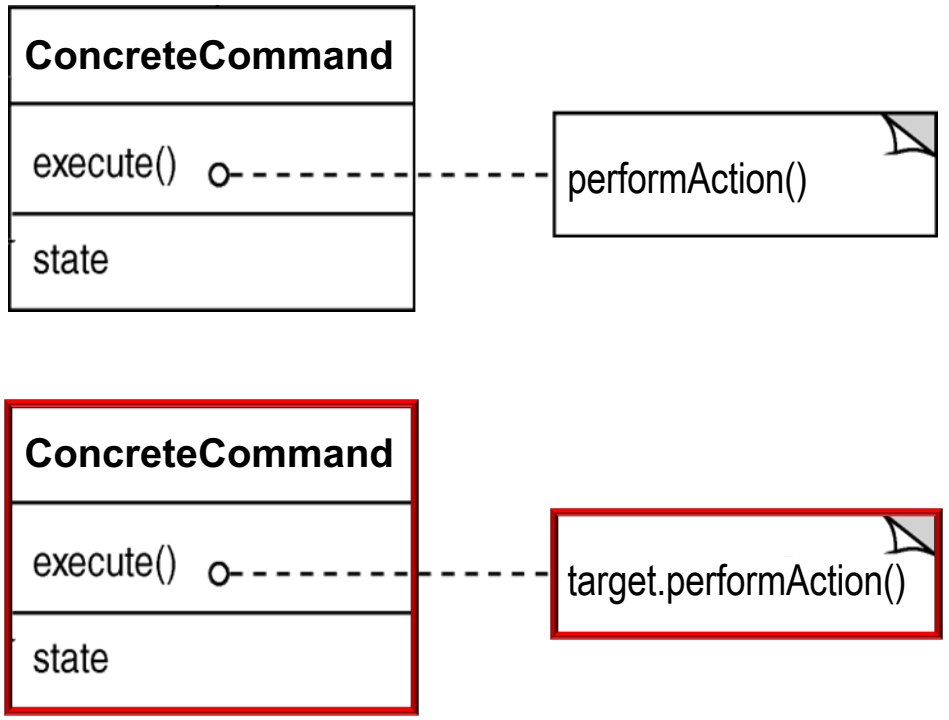
Consequences

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



Consequences

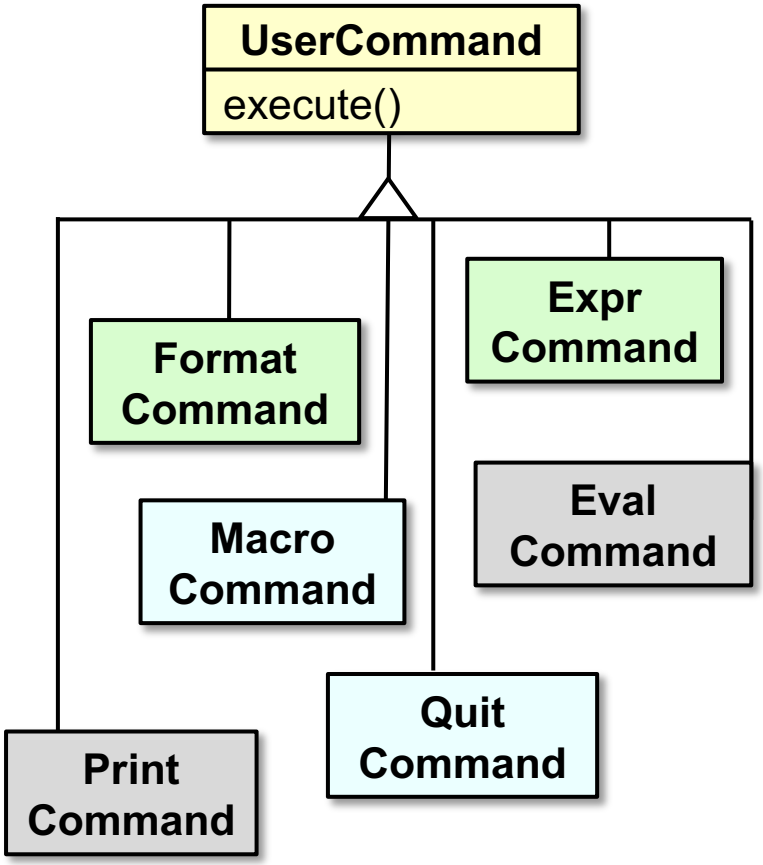
+ Abstracts the executor of a service
  - Makes programs more modular and flexible, e.g.,
    - Can bundle state and behavior into an object
    - Can forward behavior to other objects



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



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
  - 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);
```

Call a hook (factory) method to make a command based on user input

Consequences

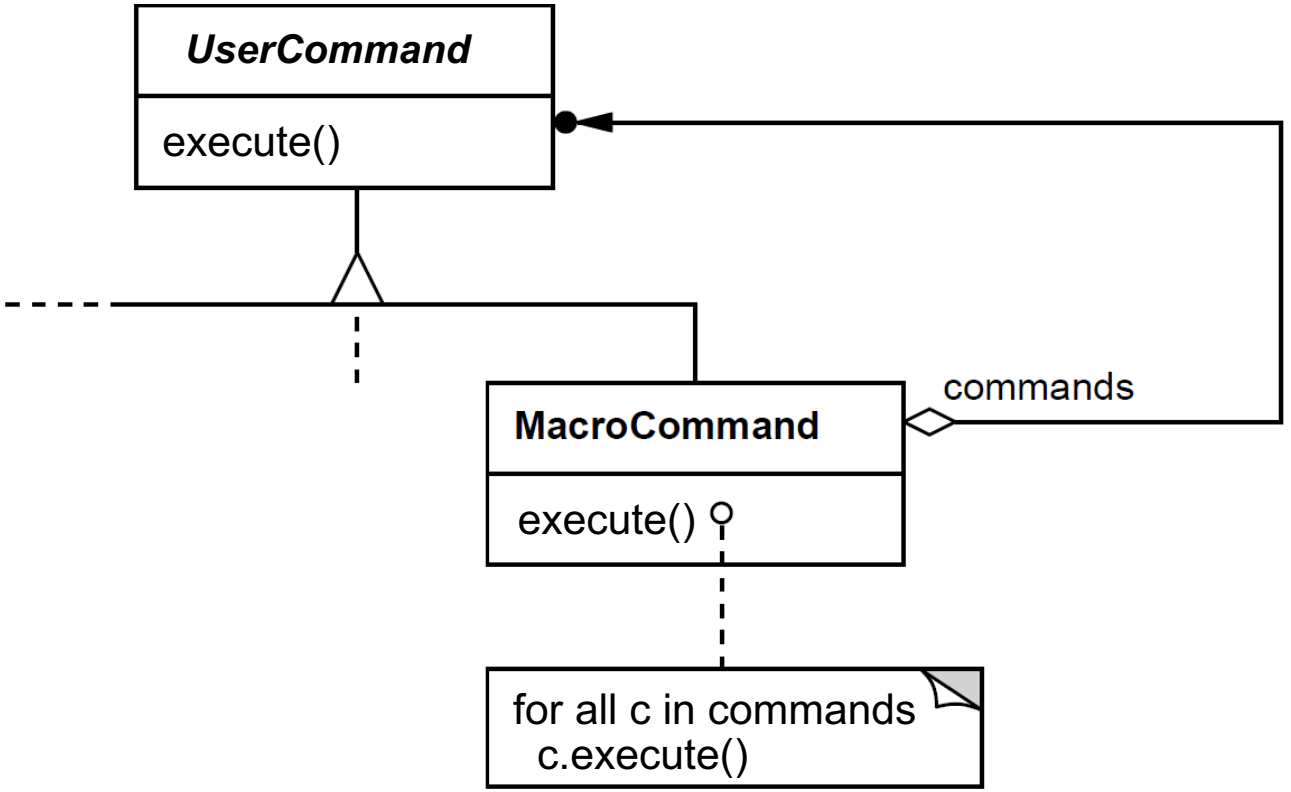
+ Abstracts the executor of a service
  - Makes programs more modular and flexible, e.g.,
    - Can bundle state and behavior into an object
    - Can forward behavior to other objects
    - Can extend behavior via subclassing
    - Can pass a command object as a parameter

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

executeCommand(command);
```

Call a hook method and pass a command to execute

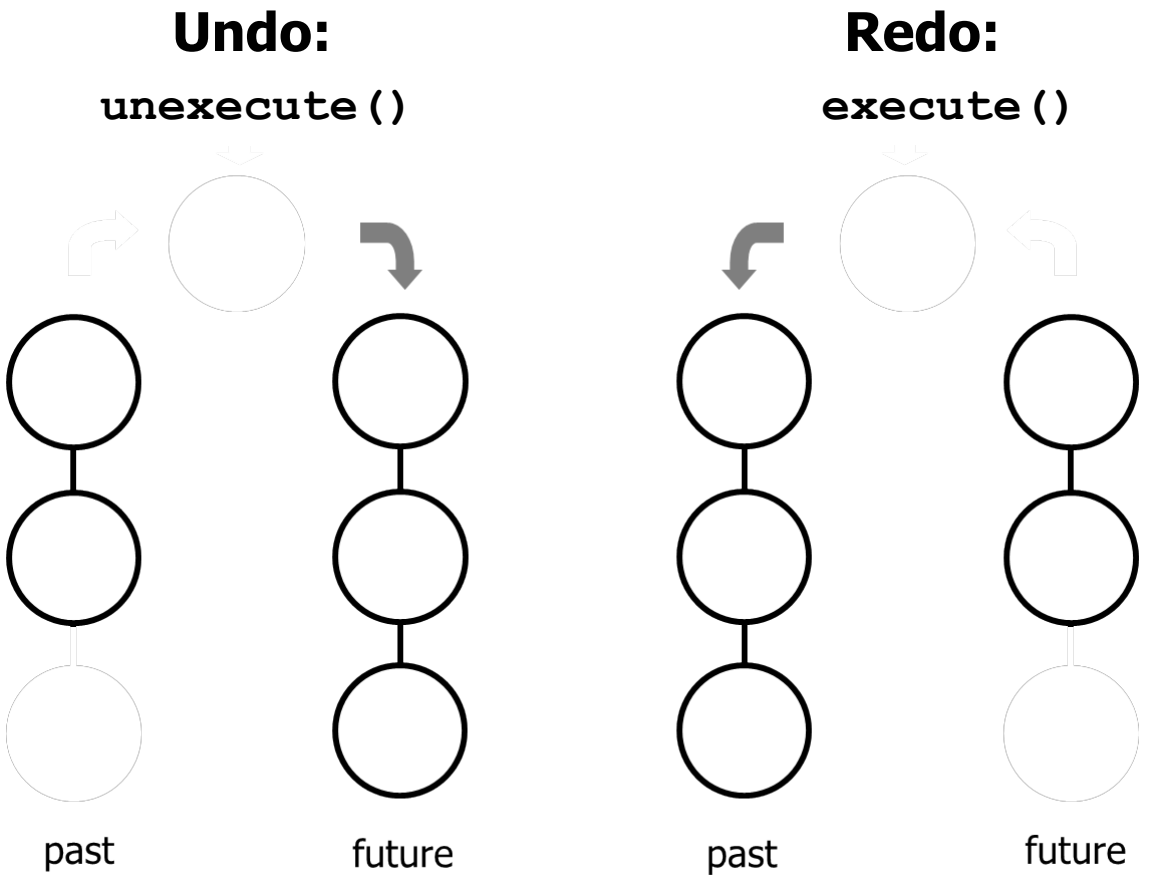
Consequences

+ Composition yields macro commands



Consequences

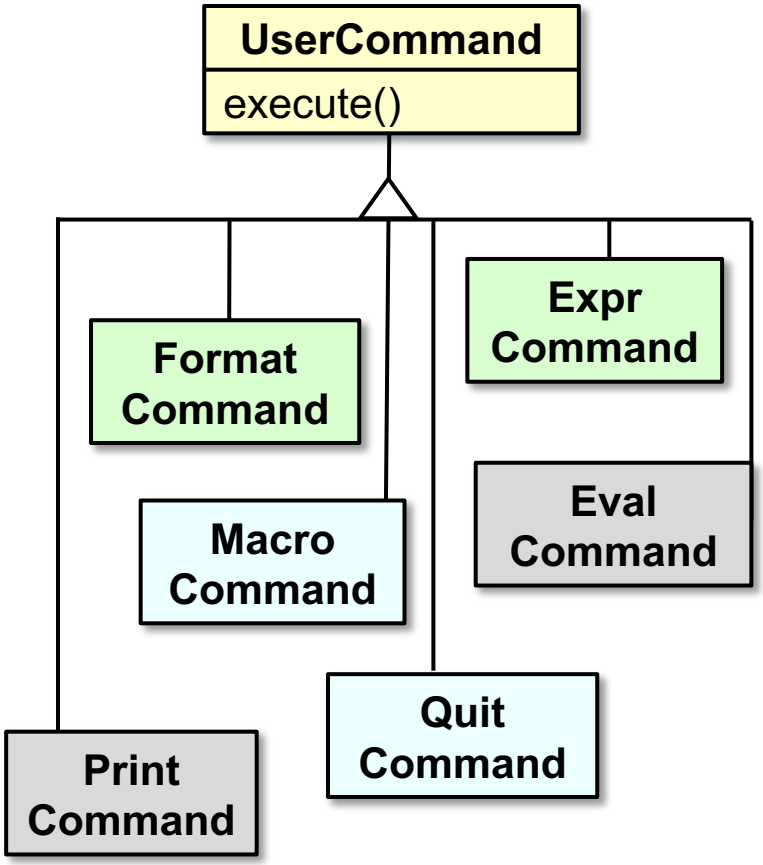
+ Supports arbitrary-level undo-redo



Case study doesn't use **`unexecute()`**, but it's a common *Command* feature.

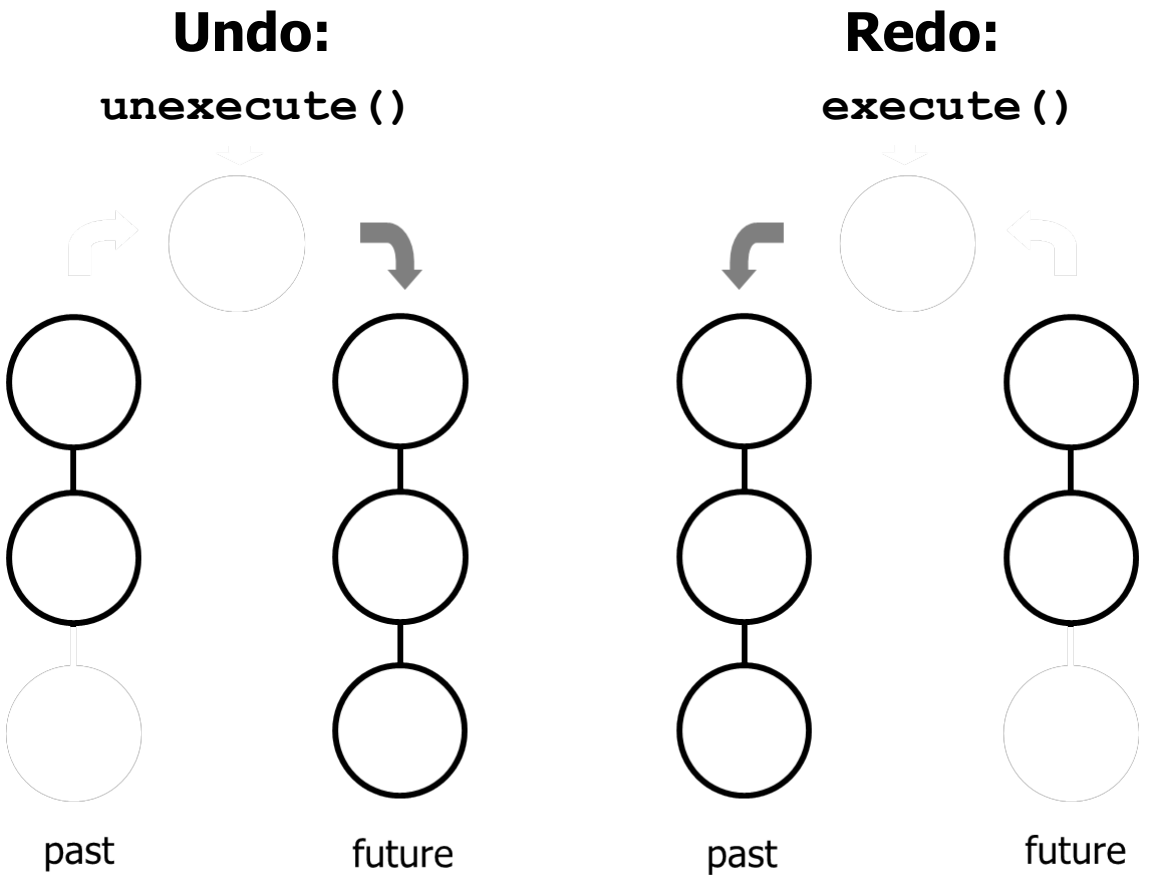
Consequences

– Might result in lots of trivial command subclasses



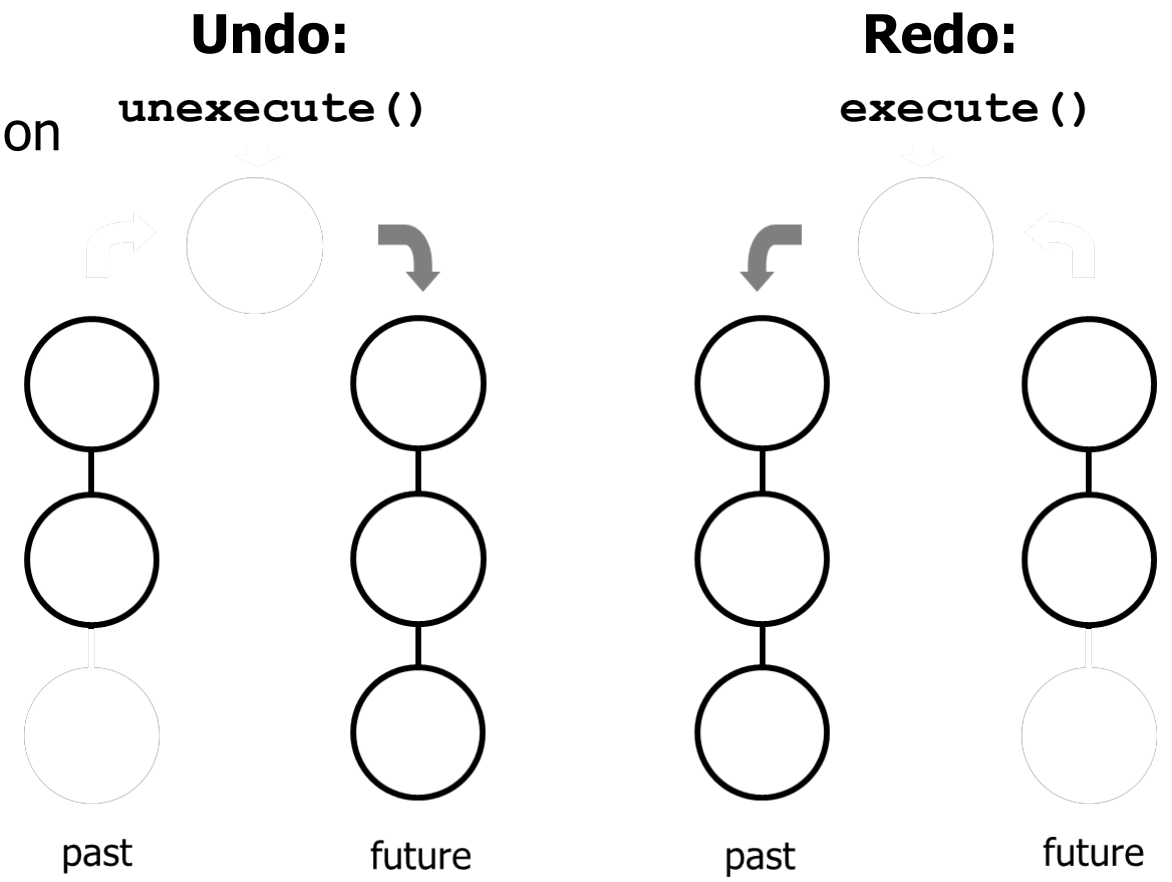
Consequences

– Excessive memory may be needed to support undo/redo operations



Implementation considerations

- Copying a command before putting it on a history list
- Avoiding error accumulation during undo/redo
- Supporting transactions



Known uses

- InterViews Actions
- MacApp, Unidraw Commands
- JDK's UndoableEdit, AccessibleAction
- GNU Emacs
- Microsoft Office tools
- Java **`Runnable`** interface

java.lang

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState, FutureTask, RenderableImageProducer, SwingWorker, Thread, TimerTask

---

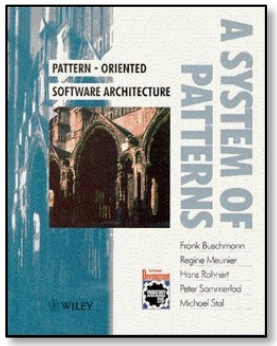
`public interface` **`Runnable`**

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 docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

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>

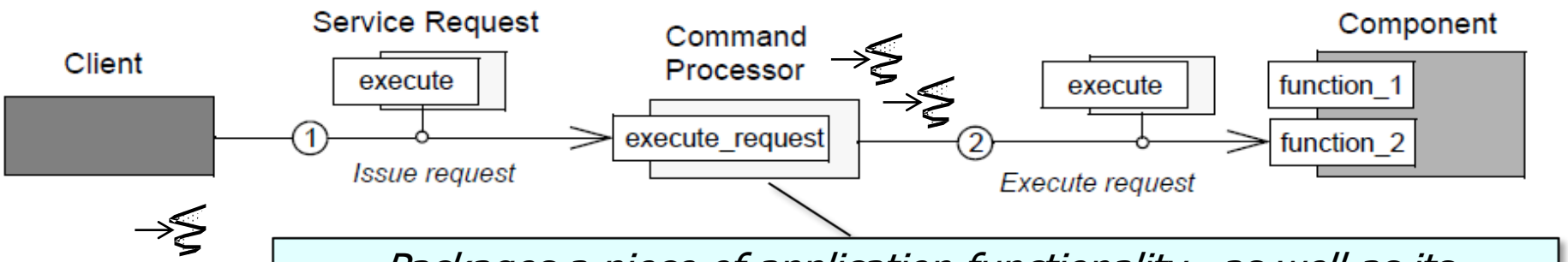
All Known Implementing Classes:

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`.

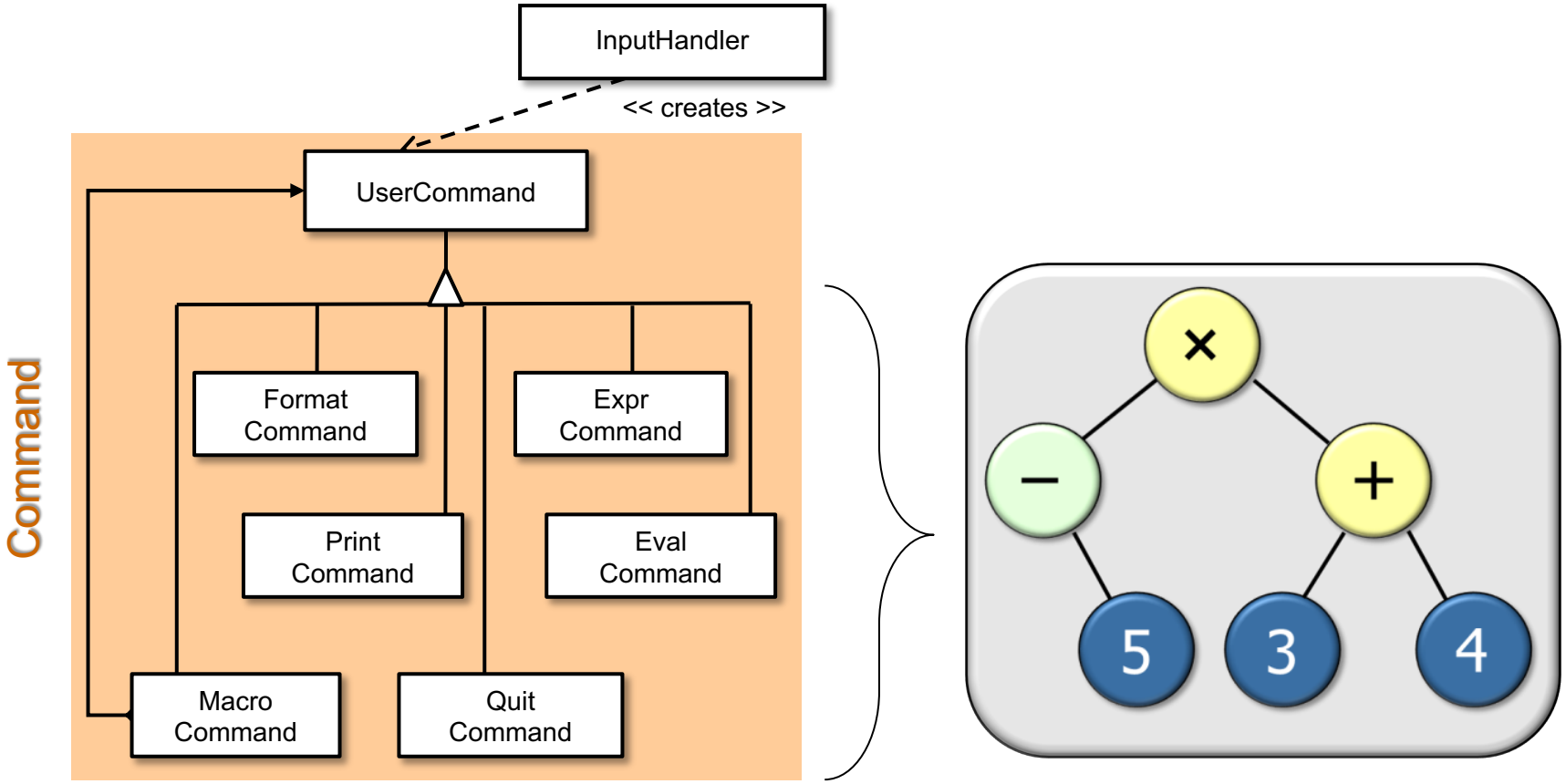


Packages a piece of application functionality—as well as its parameterization in an object—to make it usable in another context

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

Summary of the Command Pattern

- *Command* ensures users interact with the expression tree processing app in a consistent and extensible manner.



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