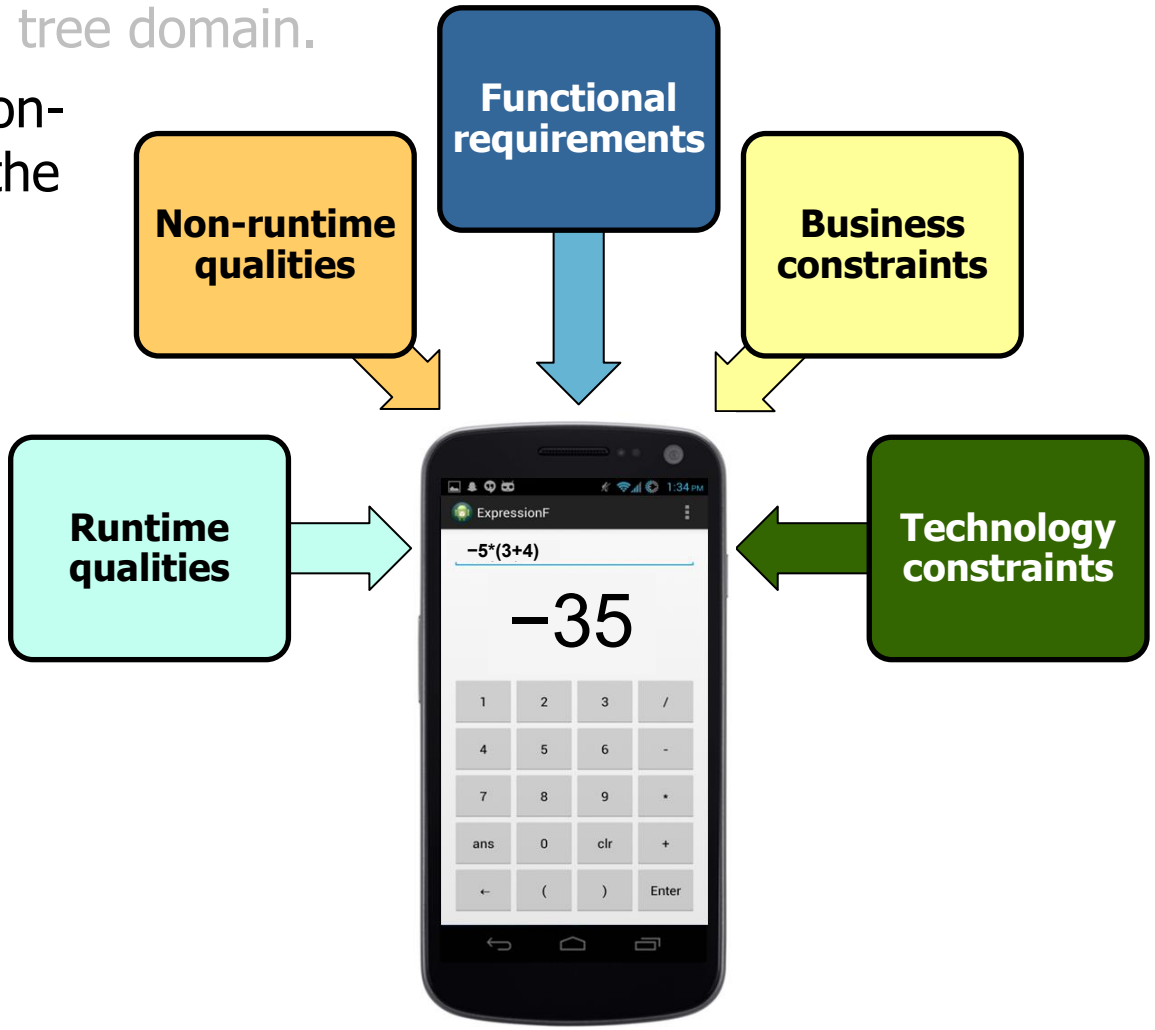
Overview of the Expression Tree Processing App Case Study (Part 2)

Douglas C. Schmidt

Learning Objectives in This Lesson

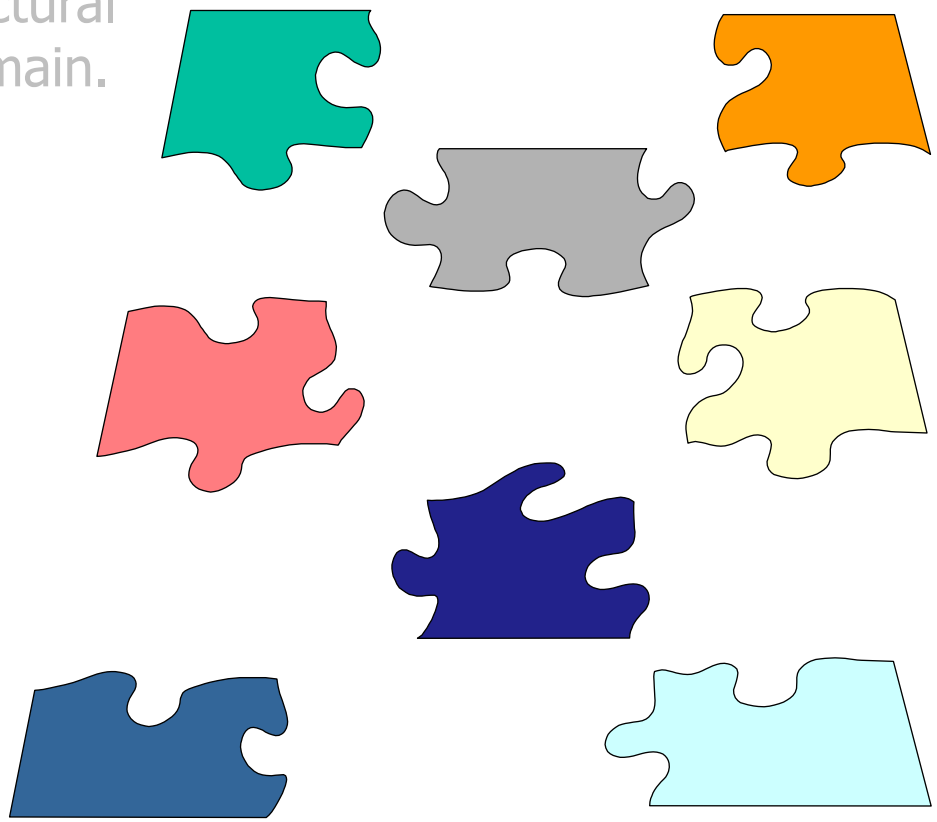
- Understand the goals of the object-oriented (OO) expression tree case study.
- Recognize the key behavioral & structural properties in the expression tree domain.
- Evaluate the functional & non-functional requirements of the case study.



Patterns are best applied to address requirements, rather than applied blindly!

Learning Objectives in This Lesson

- Understand the goals of the object-oriented (OO) expression tree case study.
- Recognize the key behavioral & structural properties in the expression tree domain.
- Evaluate the functional & non-functional requirements of the case study.
- Put all the pieces together.

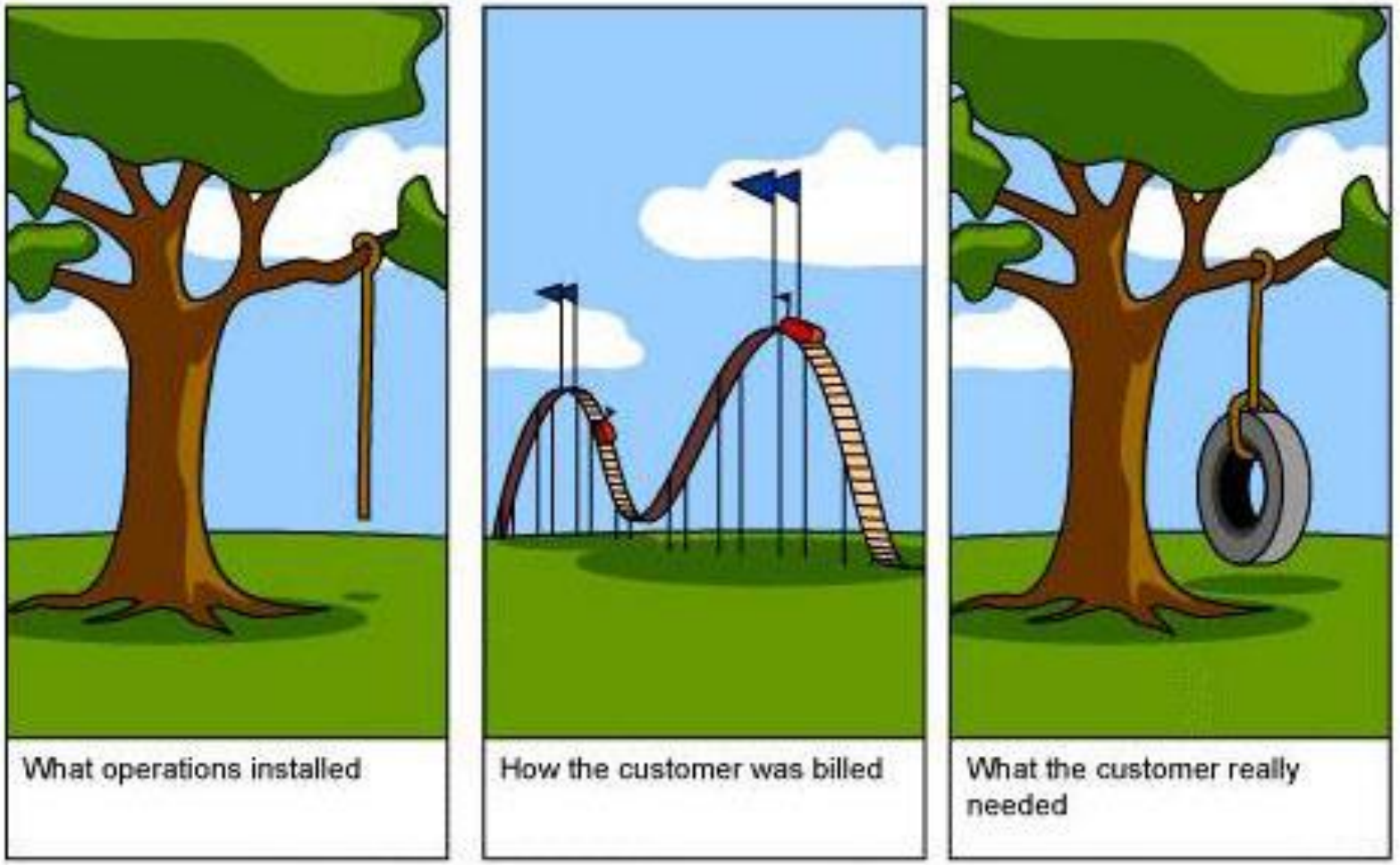


Douglas C. Schmidt

Functional & Non-Functional Requirements of the Case Study

Functional & Non-Functional Requirements

- A **functional requirement** defines what a system should be able to do, i.e., the behavior it should perform.



en.wikipedia.org/wiki/Functional_requirements has more information.

Functional & Non-Functional Requirements

- A **non-functional requirement** defines specific criteria that can be used to judge the operation of a system, rather than its specific behaviors.

- Non-Functional requirements are also called ”quality attributes” of a system.

en.wikipedia.org/wiki/Non-functional_requirement has more information.

Case Study: Functional Requirements

- "Succinct mode"—the calculator interface evaluates arithmetic expressions input by a user that must conform to a grammar

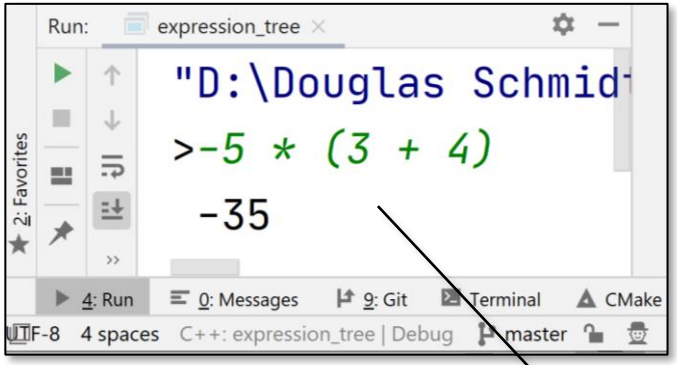
```
expr ::= factor expr-tail

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

factor ::= term factor-tail

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

mul_div ::= '×' | '/'

add_sub ::= '+' | '-`

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

Case Study: Functional Requirements

- The succinct mode can be command-line or GUI interface.
  - In the GUI version, a user presses Android buttons to enter expressions.

Case Study: Functional Requirements

- The succinct mode can be command-line or GUI interface.
  - In the GUI version, a user presses Android buttons to enter expressions.
  - In the command-line version a user designates input expressions via various notations.
    - e.g., in-fix, post-fix, etc.



In-fix expressions can contain parenthesized sub-expressions.

Case Study: Functional Requirements

- "Verbose mode"—prompts the user to enter command requests that control app behavior
  - The order of these command requests must follow a specific protocol.

Command	Behavior
format	Allows the user to select the input format (e.g., in-fix, post-fix, etc.)
expr	Allows the user to designate the current input expression
set	Sets a variable that can be used in an expression
print	Prints the current input expression using the designated traversal order (e.g., in-order, post-order, pre-order, etc.)
eval	Evaluates the value of the current input expression
quit	Exits the program

Case Study: Functional Requirements

- "Verbose mode"—prompts the user to enter command requests that control app behavior
  - The order of these command requests must follow a specific protocol.

Command	Behavior
format	Allows the user to select the input format (e.g., in-fix, post-fix, etc.)
expr	Allows the user to designate the current input expression
set	Sets a variable that can be used in an expression
print	Prints the current input expression using the designated traversal order (e.g., in-order, post-order, pre-order, etc.)
eval	Evaluates the value of the current input expression
quit	Exits the program

Case Study: Functional Requirements

- "Verbose mode"—prompts the user to enter command requests that control app behavior
  - The order of these command requests must follow a specific protocol.

Command	Behavior
format	Allows the user to select the input format (e.g., in-fix, post-fix, etc.)
expr	Allows the user to designate the current input expression
set	Sets a variable that can be used in an expression
print	Prints the current input expression using the designated traversal order (e.g., in-order, post-order, pre-order, etc.)
eval	Evaluates the value of the current input expression
quit	Exits the program

Case Study: Functional Requirements

- "Verbose mode"—prompts the user to enter command requests that control app behavior
  - The order of these command requests must follow a specific protocol.

Command	Behavior
format	Allows the user to select the input format (e.g., in-fix, post-fix, etc.)
expr	Allows the user to designate the current input expression
set	Sets a variable that can be used in an expression
print	Prints the current input expression using the designated traversal order (e.g., in-order, post-order, pre-order, etc.)
eval	Evaluates the value of the current input expression
quit	Exits the program

Case Study: Functional Requirements

- "Verbose mode"—prompts the user to enter command requests that control app behavior
  - The order of these command requests must follow a specific protocol.

Command	Behavior
format	Allows the user to select the input format (e.g., in-fix, post-fix, etc.)
expr	Allows the user to designate the current input expression
set	Sets a variable that can be used in an expression
print	Prints the current input expression using the designated traversal order (e.g., in-order, post-order, pre-order, etc.)
eval	Evaluates the value of the current input expression
quit	Exits the program

Case Study: Functional Requirements

- "Verbose mode"—prompts the user to enter command requests that control app behavior
  - The order of these command requests must follow a specific protocol.

Command	Behavior
format	Allows the user to select the input format (e.g., in-fix, post-fix, etc.)
expr	Allows the user to designate the current input expression
set	Sets a variable that can be used in an expression
print	Prints the current input expression using the designated traversal order (e.g., in-order, post-order, pre-order, etc.)
eval	Evaluates the value of the current input expression
quit	Exits the program

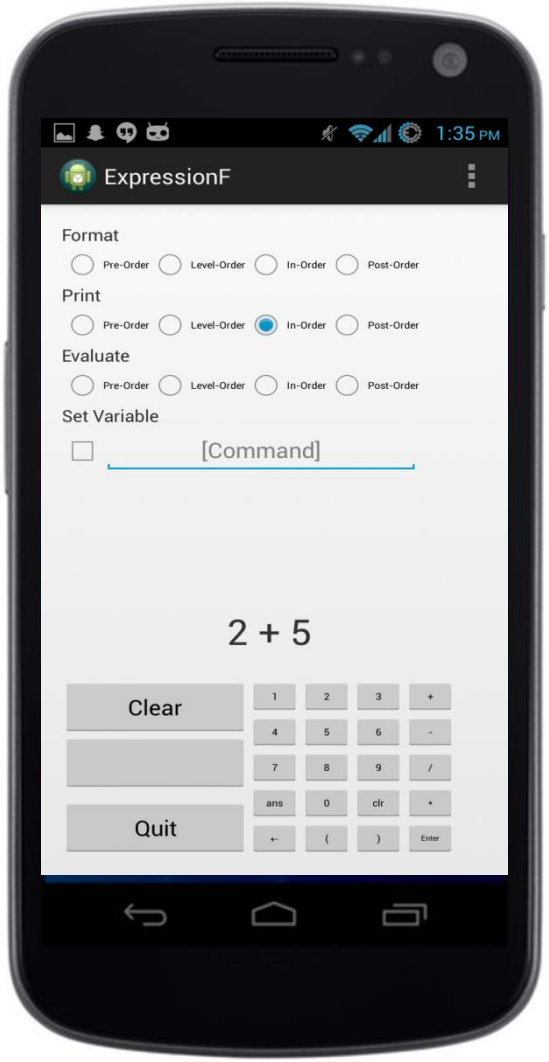
Case Study: Functional Requirements

- "Verbose mode"—prompts the user to enter command requests that control app behavior
  - The order of these command requests must follow a specific protocol.

Command	Behavior
format	Allows the user to select the input format (e.g., in-fix, post-fix, etc.)
expr	Allows the user to designate the current input expression
set	Sets a variable that can be used in an expression
print	Prints the current input expression using the designated traversal order (e.g., in-order, post-order, pre-order, etc.)
eval	Evaluates the value of the current input expression
quit	Exits the program

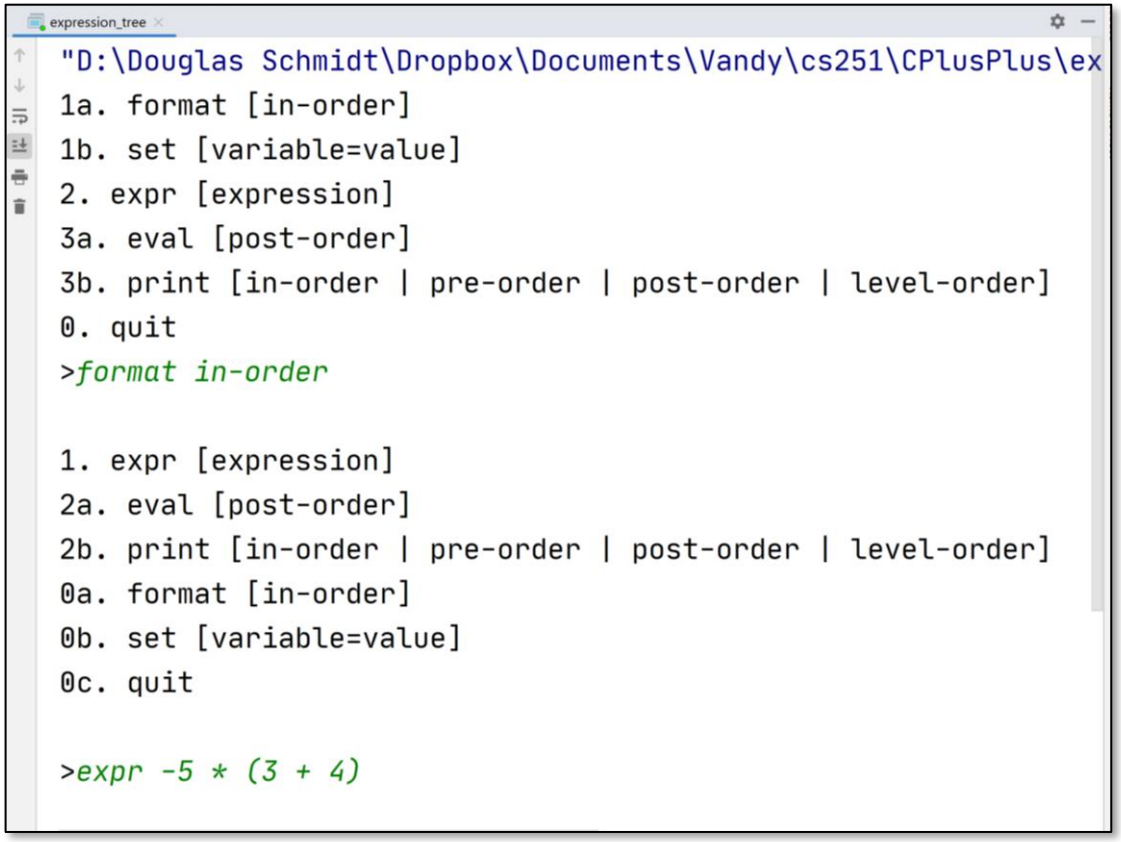
Case Study: Functional Requirements

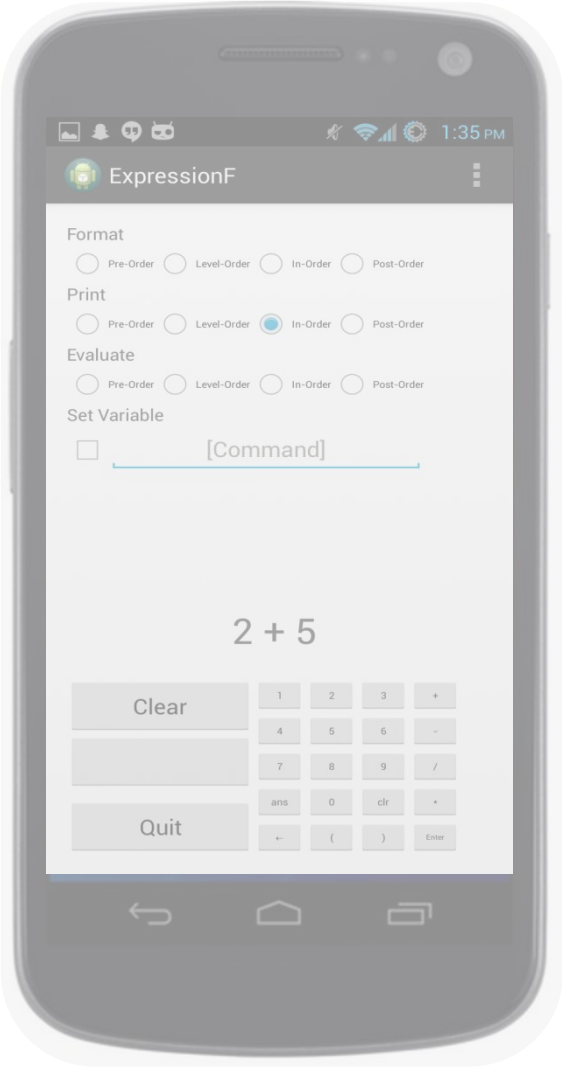
- The verbose mode can be accessed via:
  - A GUI interface



Case Study: Functional Requirements

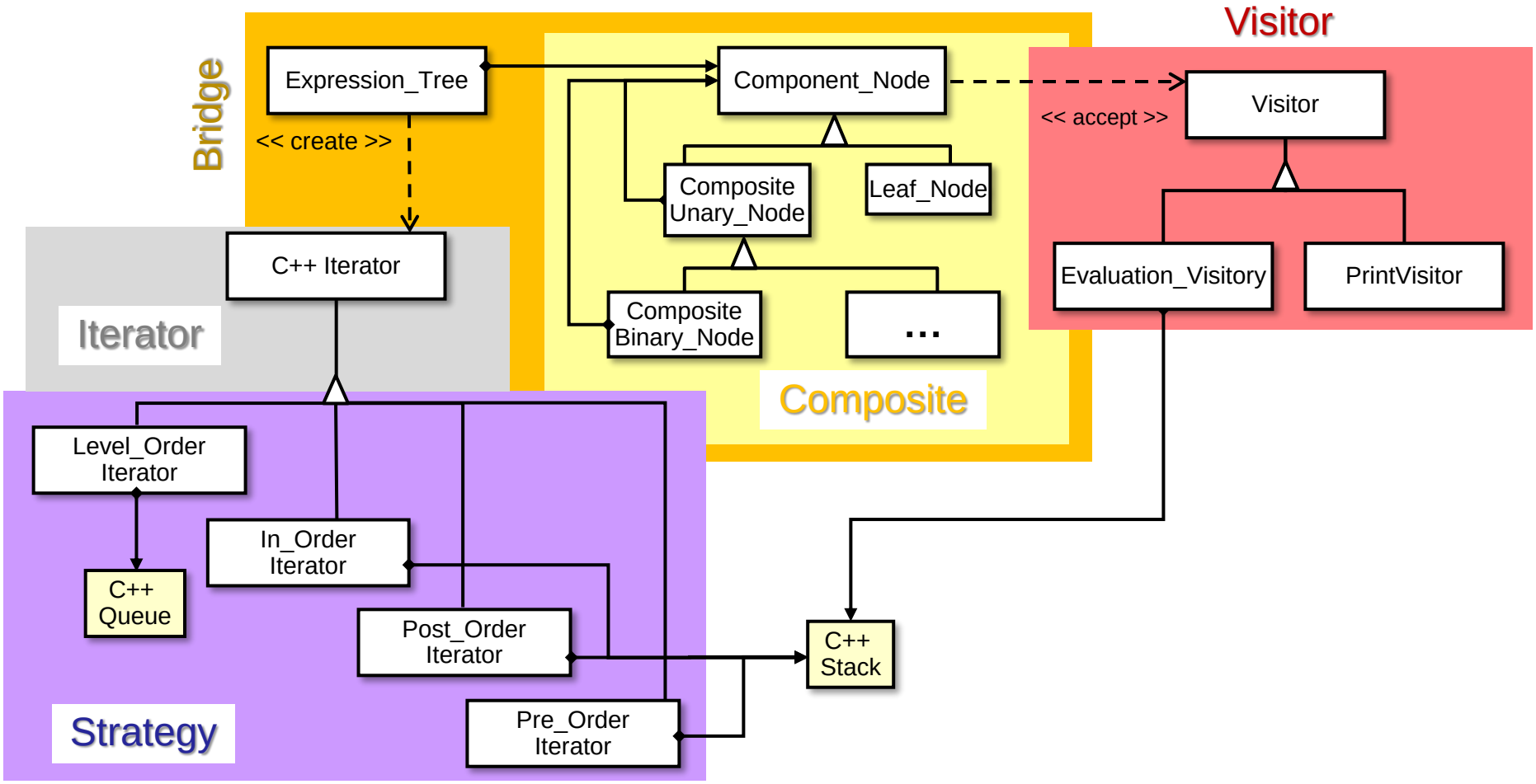
- The verbose mode can be accessed via:
  - A GUI interface
  - A command-line interface





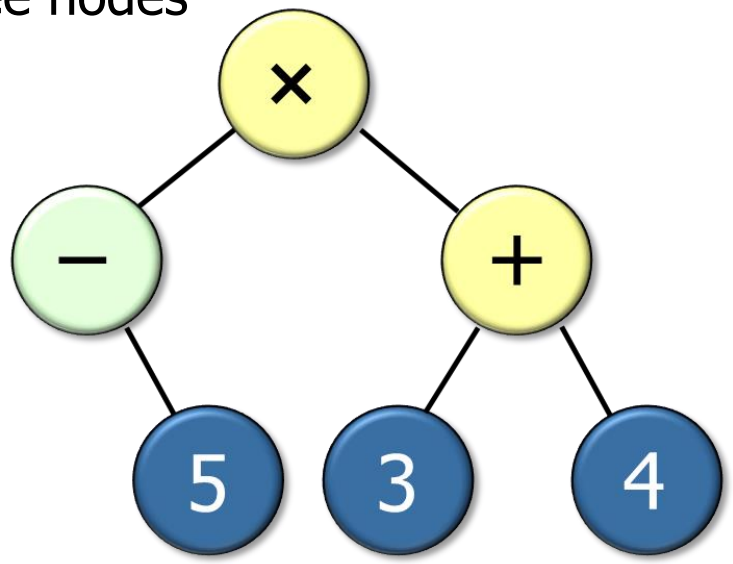
Case Study: Non-Functional Requirements

- Apply a pattern-oriented OO design to simplify extensibility & portability



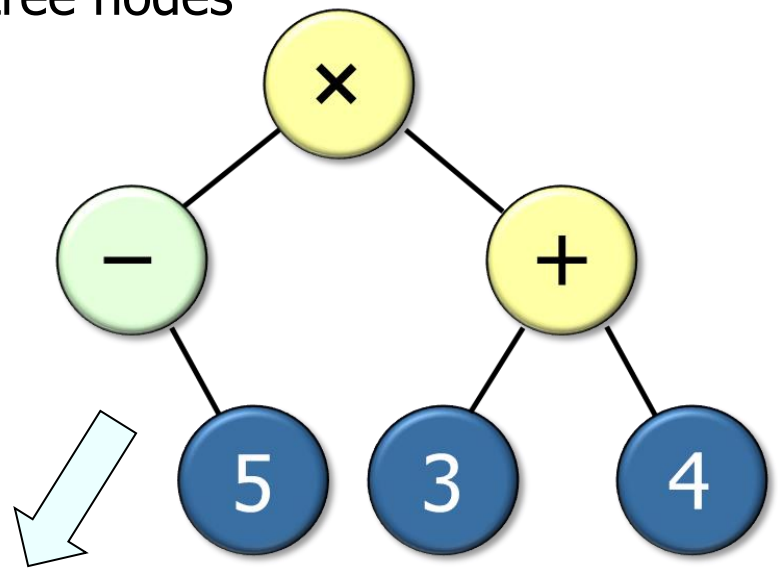
Case Study: Non-Functional Requirements

- Apply a pattern-oriented OO design to simplify extensibility & portability, e.g.,
  - Add new operations on the expression tree nodes without modifying the tree structure or implementation



Case Study: Non-Functional Requirements

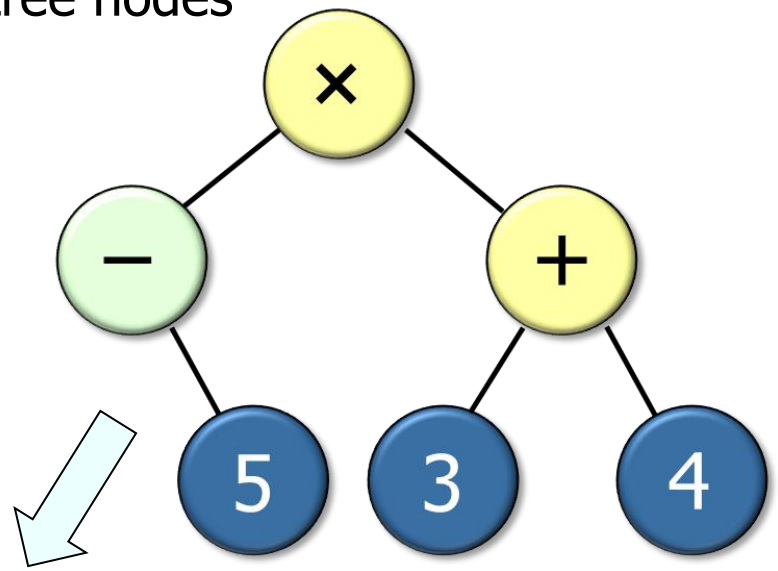
- Apply a pattern-oriented OO design to simplify extensibility & portability, e.g.,
  - Add new operations on the expression tree nodes without modifying the tree structure or implementation, e.g.,
    - Print the contents of the expression tree in various traversal orders



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

Case Study: Non-Functional Requirements

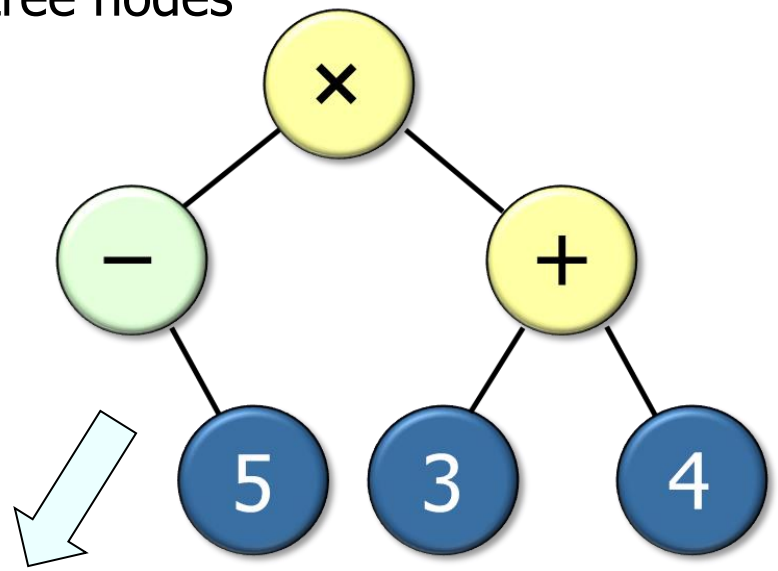
- Apply a pattern-oriented OO design to simplify extensibility & portability, e.g.,
  - Add new operations on the expression tree nodes without modifying the tree structure or implementation, e.g.,
    - Print the contents of the expression tree in various traversal orders
    - Compute the "value" of the expression tree
      - e.g., via a post-order traversal & stack-based evaluator



1. S = [5] `push(node.item())`
2. S = [-5] `push(-pop())`
3. S = [-5, 3] `push(node.item())`
4. S = [-5, 3, 4] `push(node.item())`
5. S = [-5, 7] `push(pop()+pop())`
6. S = [-35] `push(pop()*pop())`

Case Study: Non-Functional Requirements

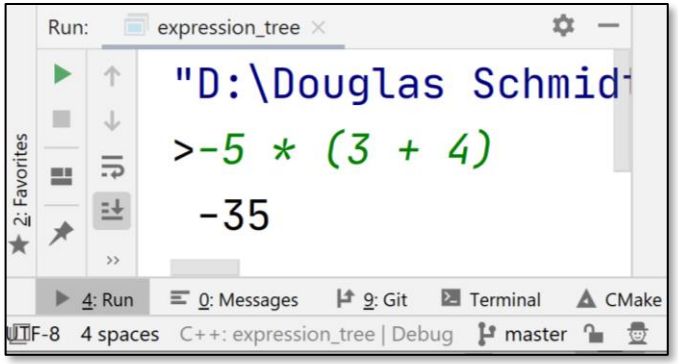
- Apply a pattern-oriented OO design to simplify extensibility & portability, e.g.,
  - Add new operations on the expression tree nodes without modifying the tree structure or implementation, e.g.,
    - Print the contents of the expression tree in various traversal orders
    - Compute the "value" of the expression tree
    - Perform semantic analysis & optimization, generate code, etc.




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

Case Study: Non-Functional Requirements

- Apply a pattern-oriented OO design to simplify extensibility & portability, e.g.,
  - Add new operations on the expression tree nodes without modifying the tree structure or implementation
  - Systematically reuse the expression tree processing app code in diverse runtime platforms, e.g.,
    - The app code is reused in both Android GUI & command-line platforms



Douglas C. Schmidt

Putting All the Pieces Together

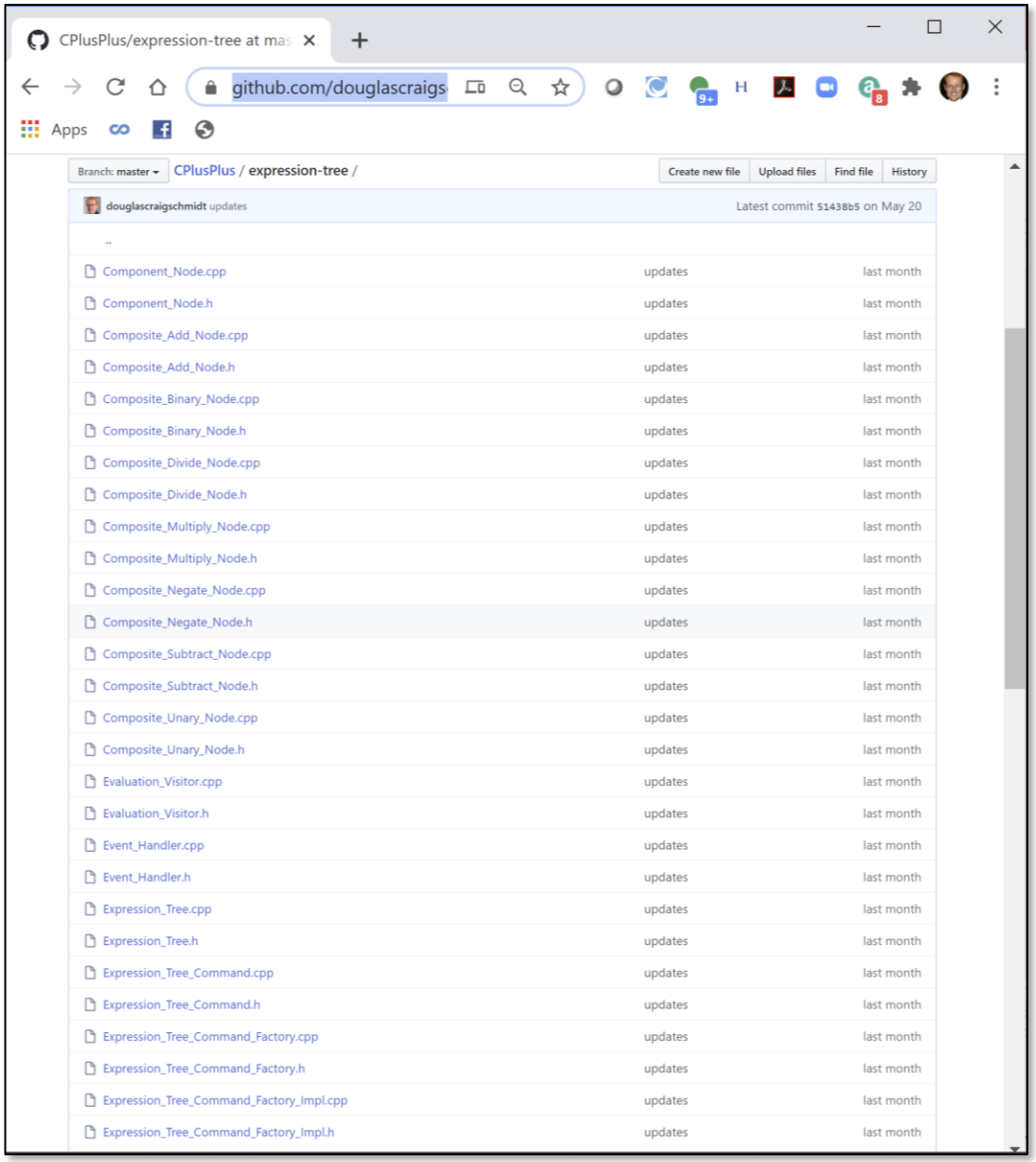
Putting All the Pieces Together

- The expression tree processing app is a realistic case study of how to apply GoF patterns.

Design Problem	Pattern
Non-extensible & error-prone designs	Composite
Minimizing impact of variability	Bridge
Inflexible expression input processing	Interpreter
Inflexible interpreter output	Builder
Scattered request implementations	Command
Inflexible creation of variabilities	Factory Method
Inflexible expression tree traversal	Iterator
Obtrusive behavior changes	Strategy
Non-extensible tree operations	Visitor
Incorrect user request ordering	State
Non-extensible operating modes	Template Method
Minimizing global variable liabilities	Singleton

Putting All the Pieces Together

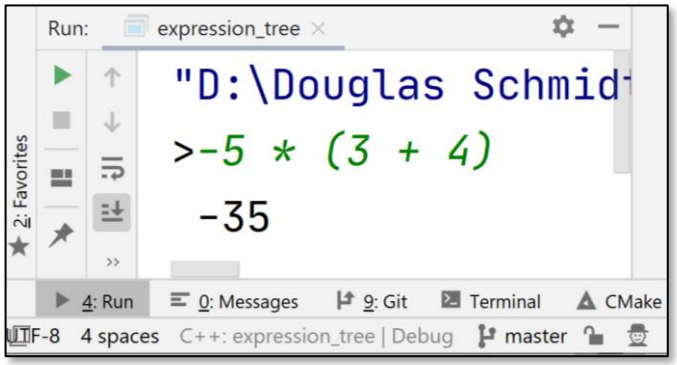
- The expression tree processing app is a realistic case study of how to apply GoF patterns.
  - All the case study code is written in C++ (~6K LOC & ~60 classes).



See github.com/douglascraigschmidt/CPlusPlus/tree/master/expression-tree

Putting All the Pieces Together

- The expression tree processing app is a realistic case study of how to apply GoF patterns.
  - All the case study code is written in C++.
  - There are command-line & Android GUI-based versions.



See github.com/douglascraigschmidt/CPlusPlus/tree/master/expression-tree