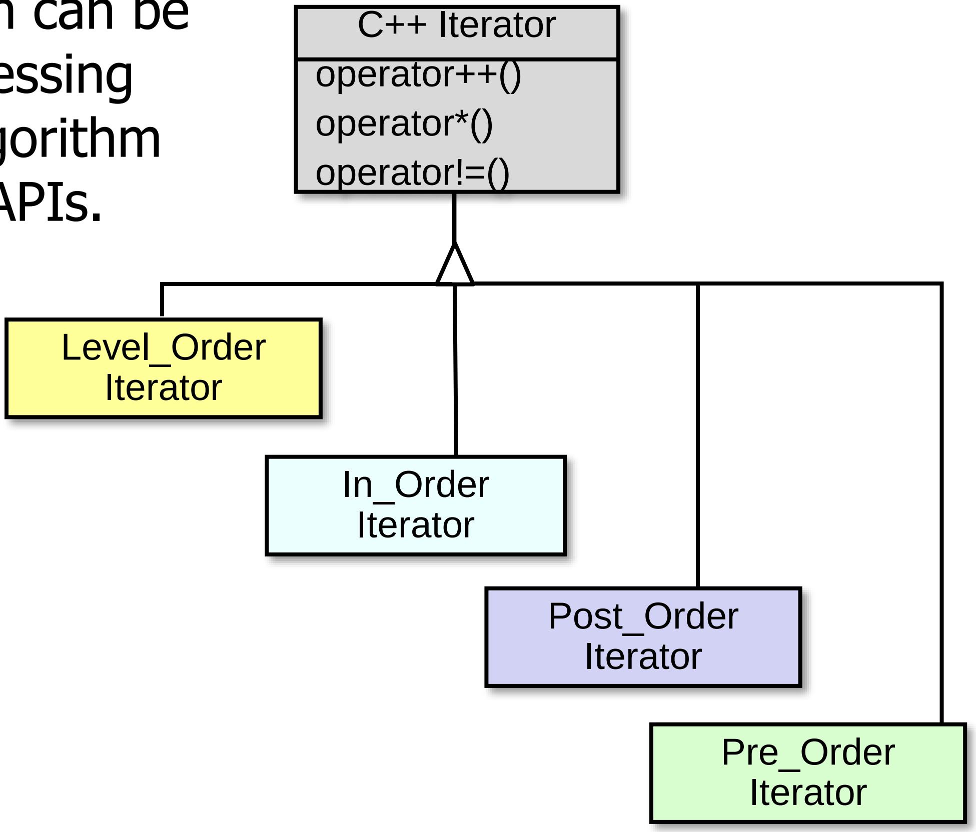
The Strategy Pattern

Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Strategy* pattern can be applied in the expression tree processing app to encapsulate variability of algorithm & platform behaviors via common APIs.

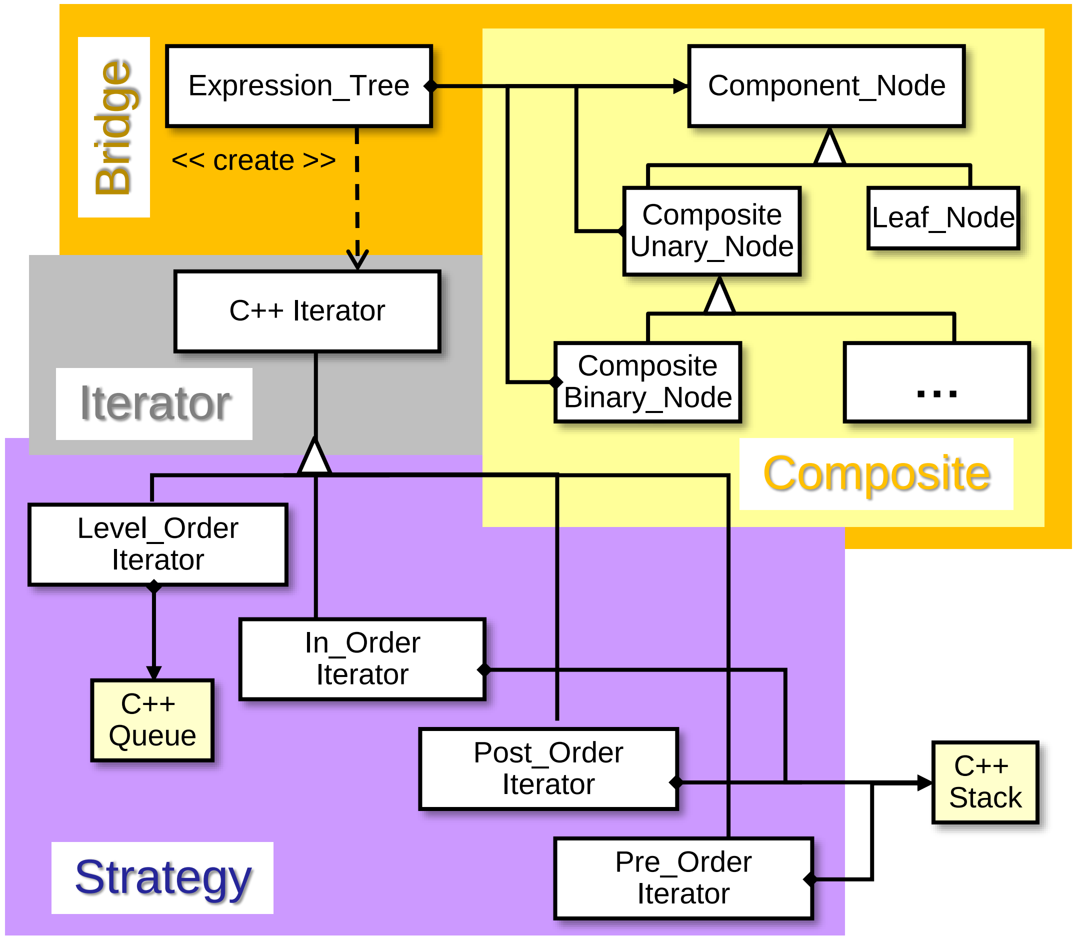


Douglas C. Schmidt

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

A Pattern for Changing Behaviors Transparently

Purpose*: Encapsulate variability of behaviors via a common API whose implementations can be changed transparently with respect to clients.*



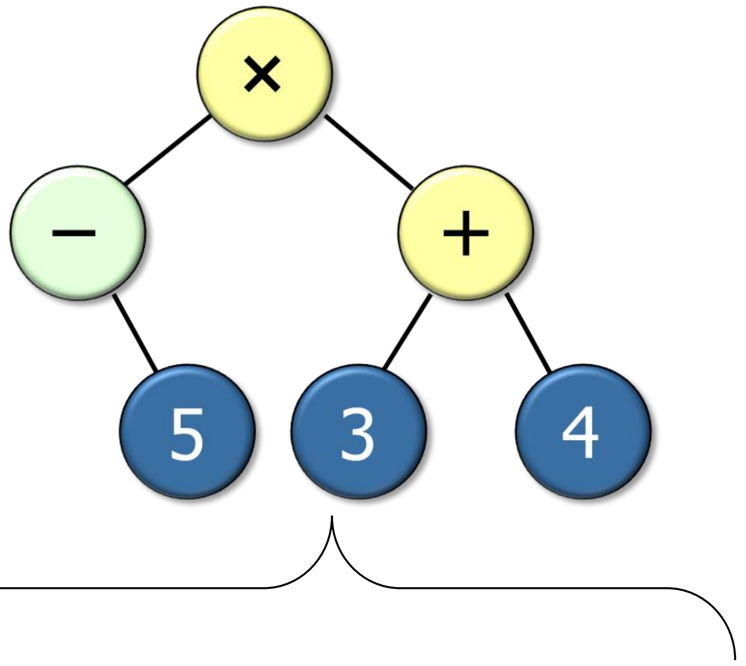
Strategy decouples the interface of a behavior from its implementations.

Context: OO Expression Tree Processing App

- Certain program behaviors must change in response to different user requests & runtime platforms

Context: OO Expression Tree Processing App

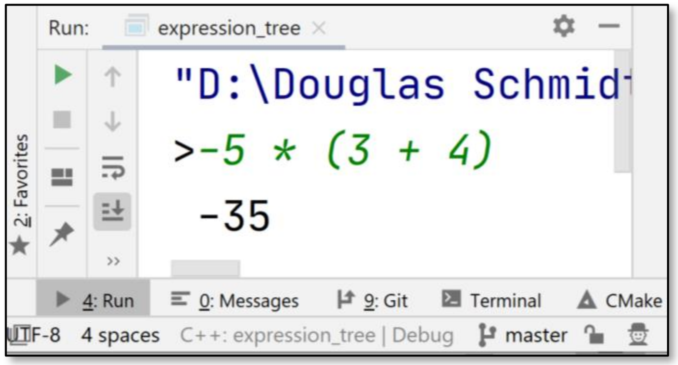
- Certain program behaviors must change in response to different user requests & runtime platforms, e.g.,
  - Different algorithms are needed to traverse the expression tree in different orders.
    - e.g., to print & evaluate the tree



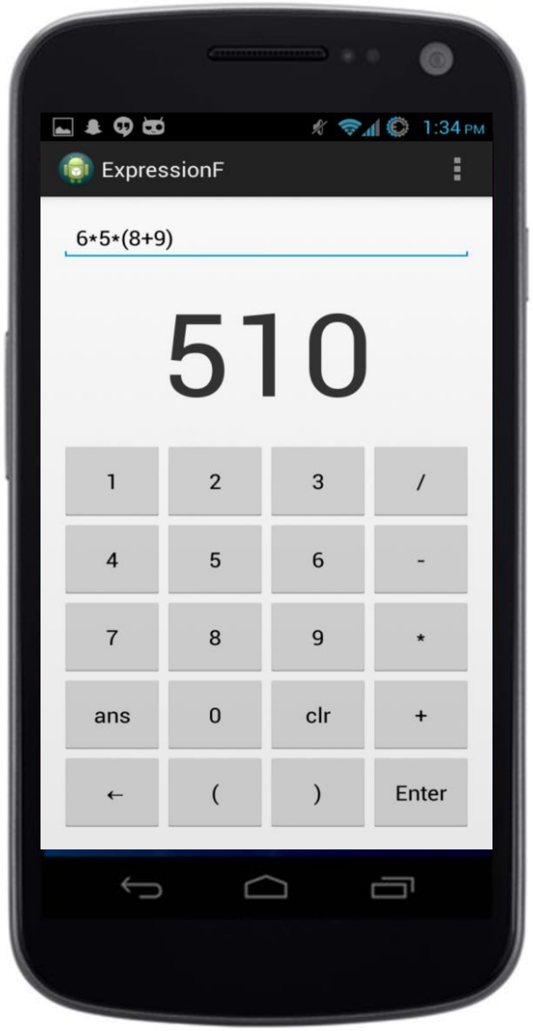
- "In-order" traversal = **−5×(3+4)**
- "Pre-order" traversal = **×−5+34**
- "Post-order" traversal = **5−34+×**
- "Level-order" traversal = **×−+534**

Context: OO Expression Tree Processing App

- Certain program behaviors must change in response to different user requests & runtime platforms, e.g.,
  - Different algorithms are needed to traverse the expression tree in different orders.
  - Different input & output mechanisms are needed in different runtime platforms.
    - e.g., Android GUI & command-line platforms

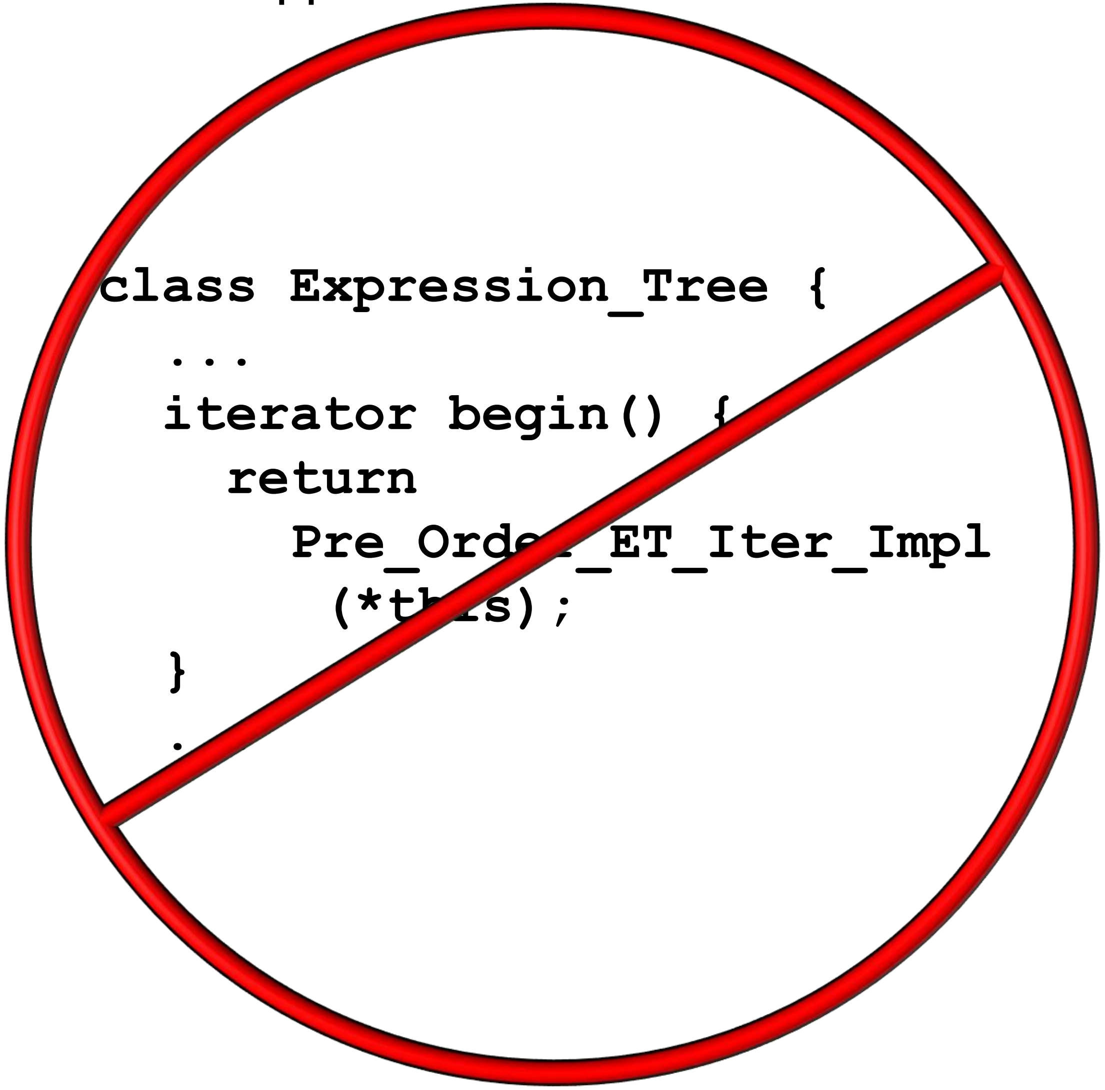


```
"D:\Douglas Schmid
>-5 * (3 + 4)
 -35
```



Problem: Obtrusive Behavior Changes

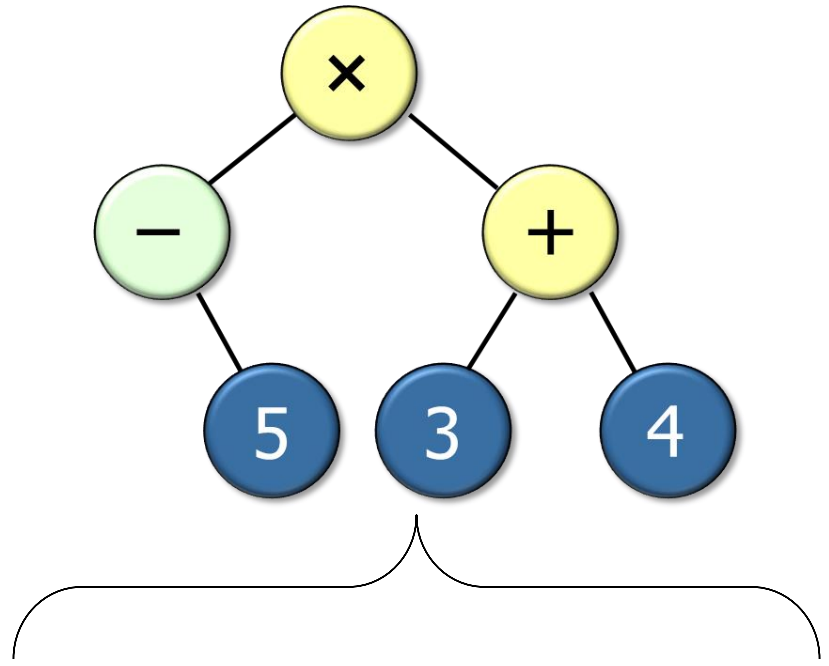
- Hard-coding certain implementations of these behaviors is problematic since obtrusive changes would be needed to support alternatives



```
class Expression_Tree {
  ...
  iterator begin() {
    return
      Pre_Order_ET_Iter_Impl
        (*this);
  }
  .
```

Problem: Obtrusive Behavior Changes

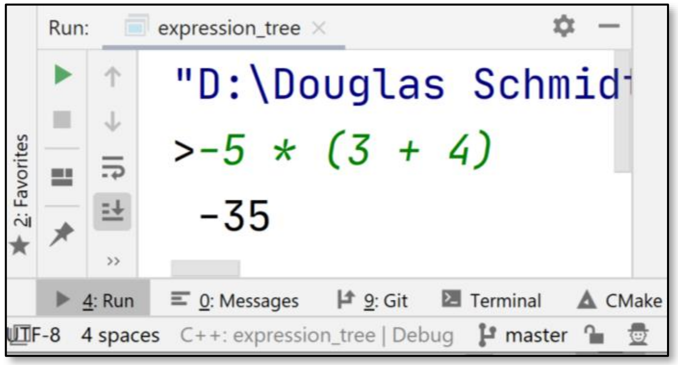
- Hard-coding certain implementations of these behaviors is problematic since obtrusive changes would be needed to support alternatives, e.g.,
  - Adding new traversal algorithms



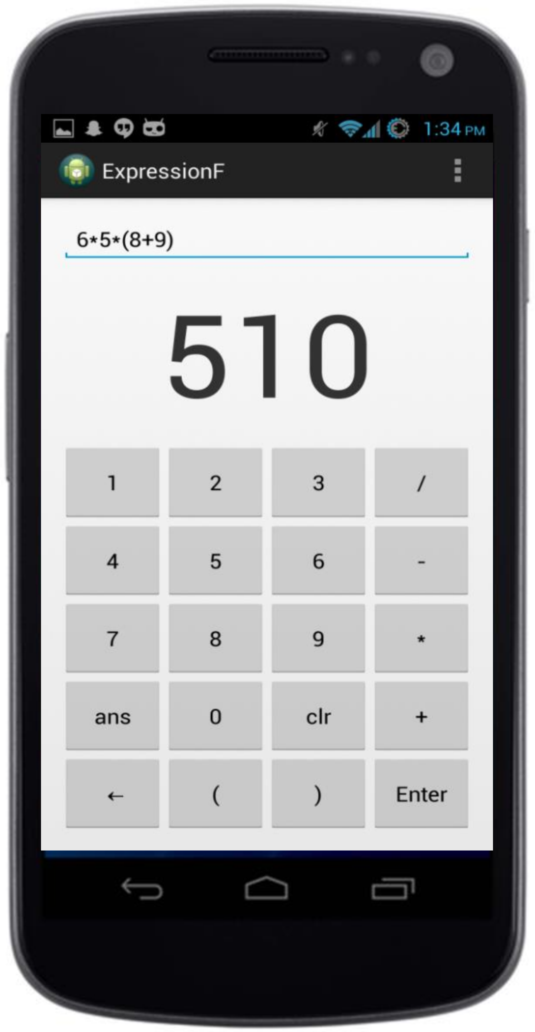
- "In-order" traversal = **–5×(3+4)**
- "Pre-order" traversal = **×–5+34**
- "Post-order" traversal = **5–34+×**
- "Level-order" traversal = **×–+534**

Problem: Obtrusive Behavior Changes

- Hard-coding certain implementations of these behaviors is problematic since obtrusive changes would be needed to support alternatives, e.g.,
  - Adding new traversal algorithms
  - Supporting different runtime platforms



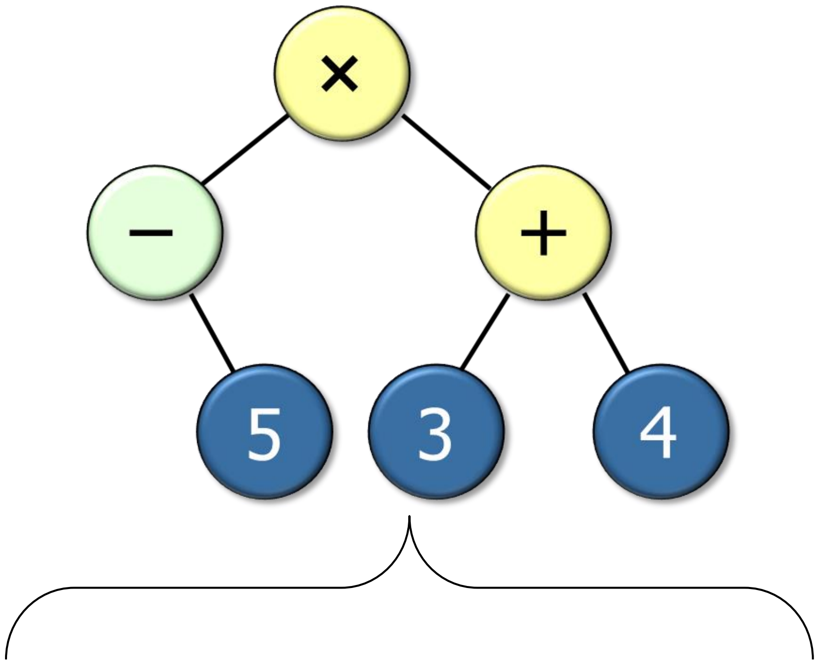
```
"D:\Douglas Schmid
>-5 * (3 + 4)
 -35
```



```
6*5*(8+9)
510
```

Solution: Create an Abstraction to Select Behaviors

- Define a family of behaviors.
  - e.g., algorithms for traversing an expression tree in various orders



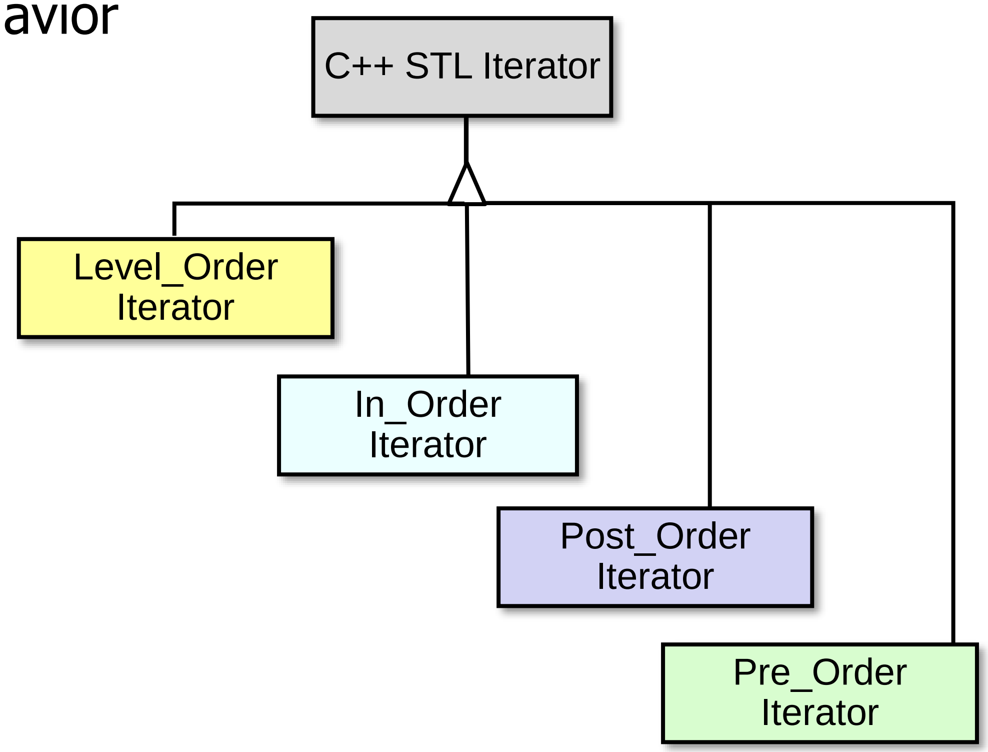
- "In-order" traversal = **−5×(3+4)**
- "Pre-order" traversal = **×−5+34**
- "Post-order" traversal = **5−34+×**
- "Level-order" traversal = **×−+534**

Solution: Create an Abstraction to Select Behaviors

- Encapsulate all behaviors to have a common API.
  - e.g., the C++ STL iterator interface

Solution: Create an Abstraction to Select Behaviors

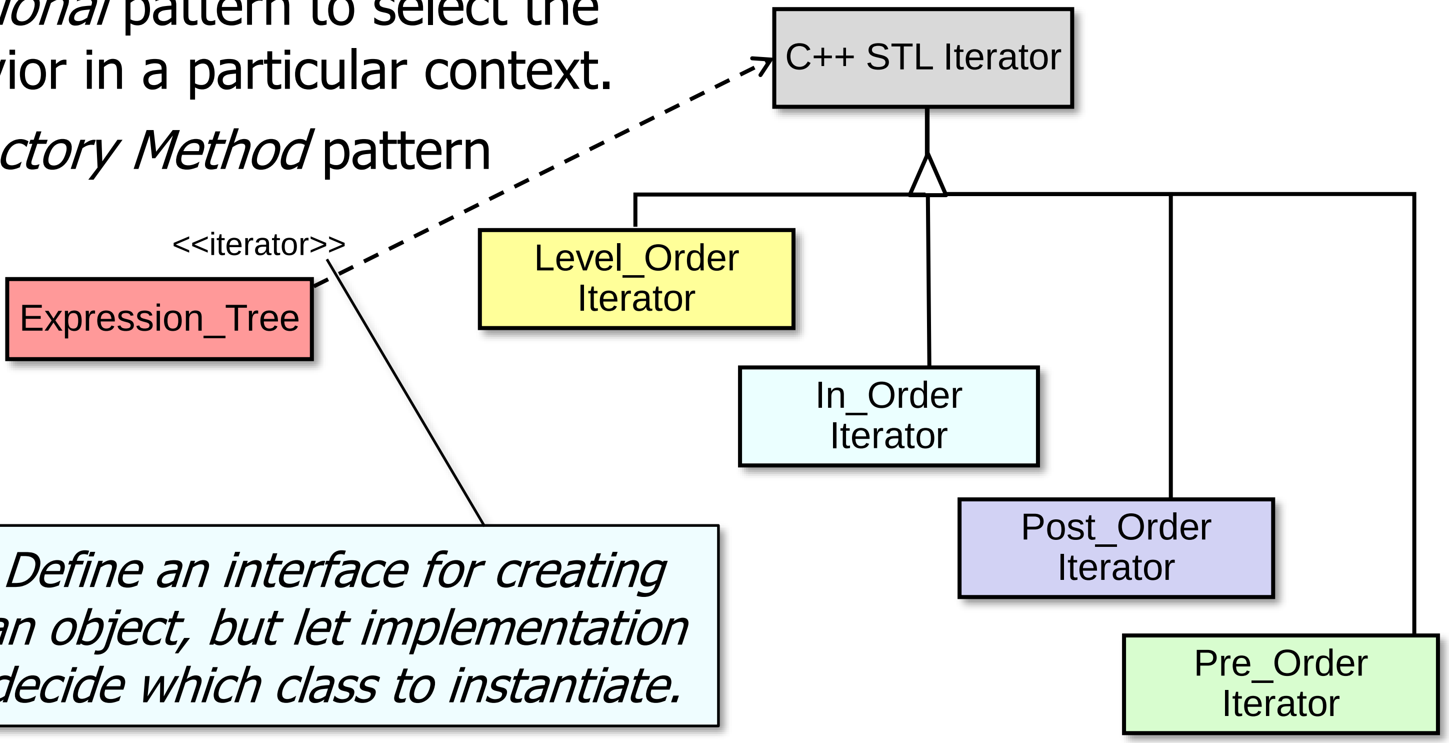
- Make implementations of the behavior interchangeable.
  - Different traversal orders all implement the same C++ STL iterator interface



Strategy encapsulates multiple traversal algorithms via a common API.

Solution: Create an Abstraction to Select Behaviors

- Apply a *Creational* pattern to select the desired behavior in a particular context.
  - e.g., the *Factory Method* pattern

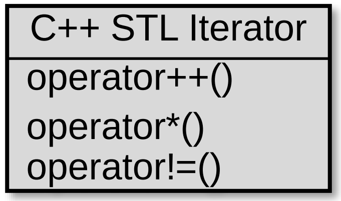


Define an interface for creating an object, but let implementation decide which class to instantiate.

See en.wikipedia.org/wiki/Factory_method_pattern

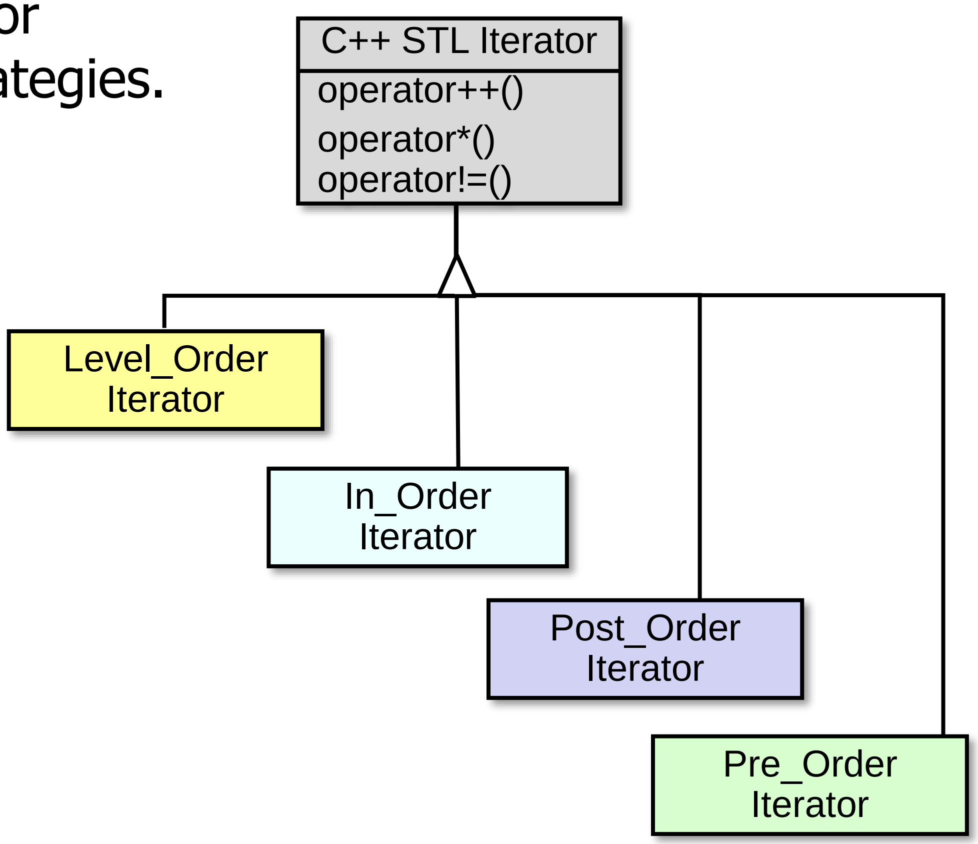
Strategy Hierarchy Overview

- The root of the hierarchy is based on the *Iterator* pattern & C++ STI iterator interface.



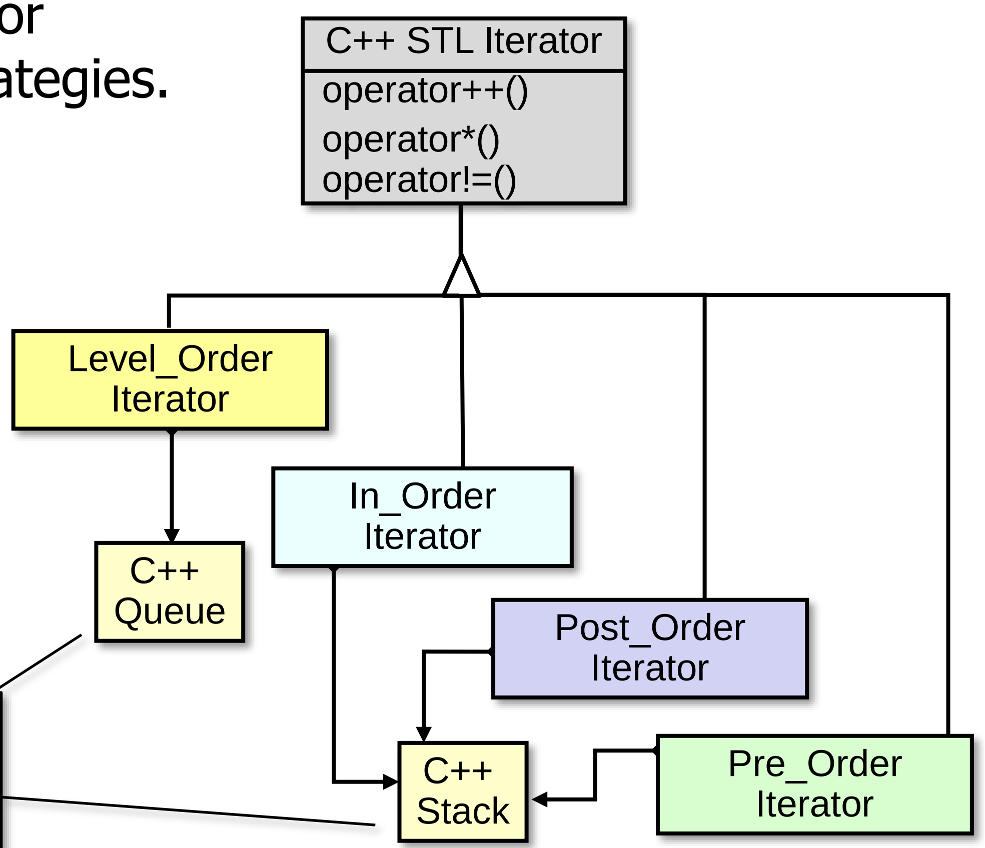
Strategy Hierarchy Overview

- Implementations of the C++ Iterator interface define various iterator strategies.
  - e.g., pre-order, post-order, level-order, & in-order iterators



Strategy Hierarchy Overview

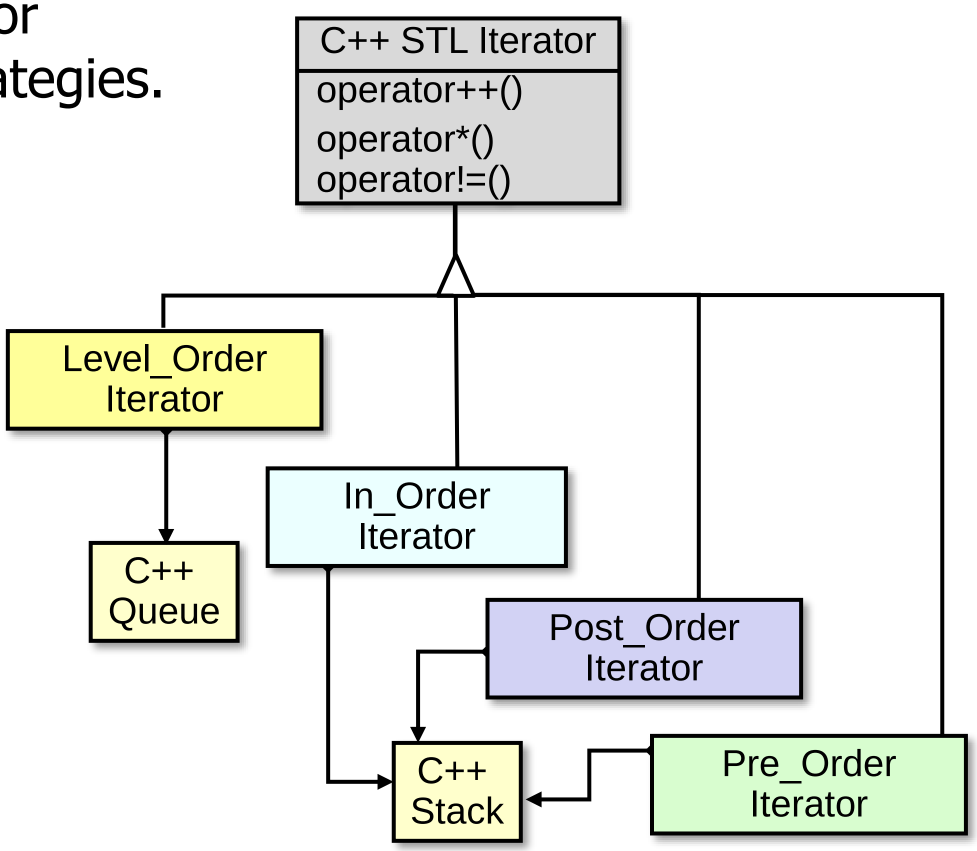
- Implementations of the C++ Iterator interface define various iterator strategies.
  - e.g., pre-order, post-order, level-order, & in-order iterators



C++ ***`stack`*** *&* ***`queue`*** *objects track the state needed to perform non-recursive tree traversals.*

Strategy Hierarchy Overview

- Implementations of the C++ Iterator interface define various iterator strategies.
  - e.g., pre-order, post-order, level-order, & in-order iterators



- **Commonality**: the C++ Iterator interface defines a common strategy API
- **Variability**: implementations of this interface define concrete strategies