

Overview of Design Patterns in Java (Day 1)

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

**Institute for Software
Integrated Systems**

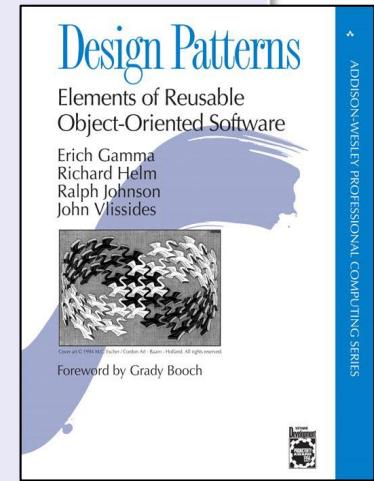
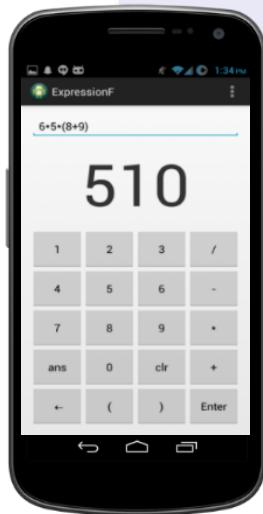
**Vanderbilt University
Nashville, Tennessee, USA**



Learning Objectives in this Lesson

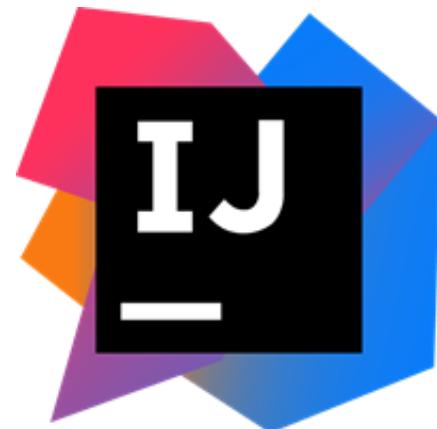
- Know what topics we'll cover

	Creational	Structural	Behavioral
Class	Factory Method ✓	Adapter ✓ (class)	Interpreter ✓ Template Method ✓
Object	Abstract Factory ✓ Builder ✓ Prototype Singleton ✓	Adapter ✓ (object) Bridge ✓ Composite ✓ Decorator ✓ Flyweight Façade Proxy	Chain of Responsibility Command ✓ Iterator ✓ Mediator Memento Observer ✓ State ✓ Strategy ✓ Visitor ✓



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java JDK/JRE & relevant IDEs



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java JDK/JRE & relevant IDEs
- Be aware of other digital learning resources



Learning Objectives in this Lesson

- Know what topics we'll cover
- Learn where to find Java JDK/JRE & relevant IDEs
- Be aware of other digital learning resources
- Be able to locate examples of Java programs

USE THE
SOURCE LIVED
CODE



Branch: master		New pull request	Create new file	Upload files	Find file	Clone or download
douglasraigschmidt	updates					Latest commit a67fd89 35 minutes ago
BarrierTaskGang		Updates				10 months ago
BuggyQueue		updates				a day ago
BusySynchronizedQueue		updates				4 months ago
DeadlockQueue		Refactored				3 years ago
ExpressionTree		Updates				10 months ago
Factorials		update				5 days ago
ImageStreamGang		updates				22 days ago
ImageTaskGangApplication		Updates				10 months ago
Java8		update				3 days ago
PalantiriManagerApplication		Updates				7 months ago
PingPongApplication		Updates				10 months ago
PingPongWrong		Refactored				3 years ago
SearchStreamForkJoin		updates				35 minutes ago
SearchStreamGang						3 hours ago
SearchStreamSpliterator						37 minutes ago
SearchTaskGang						4 months ago
SimpleAtomicLong						2 years ago
SimpleBlockingQueue						4 months ago
SimpleSearchStream						6 days ago
ThreadJoinTest		updates				22 days ago
ThreadedDownloads		Updates				10 months ago
UserOrDaemonExecutor		Refactored				3 years ago
UserOrDaemonRunnable		Updates.				2 years ago
UserOrDaemonThread		Updates				10 months ago
UserThreadInterrupted		Update				2 years ago
.gitattributes		Committed.				3 years ago
.gitignore		Updates				10 months ago
README.md		updates				21 days ago

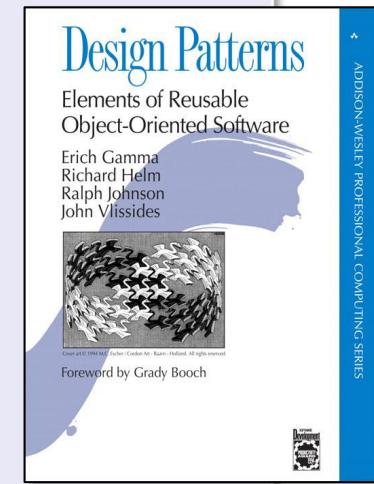
We'll just focus on the ExpressionTree example, but feel free to clone or download the entire repo!

Overview of this Course

Overview of this Course

- We focus on programming “Gang-of-Four” (GoF) design patterns in Java, e.g.

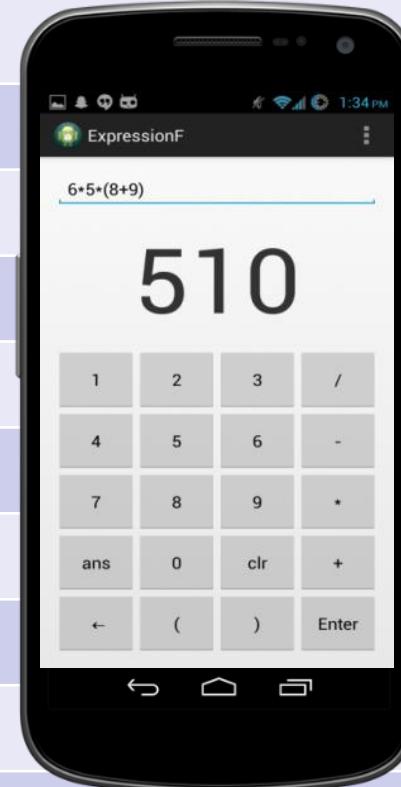
	Creational	Structural	Behavioral
Class	Factory Method ✓	Adapter ✓ (class)	Interpreter ✓ Template Method ✓
Object	Abstract Factory ✓ Builder ✓ Prototype Singleton ✓	Adapter ✓ (object) Bridge ✓ Composite ✓ Decorator ✓ Flyweight Façade Proxy	Chain of Responsibility Command ✓ Iterator ✓ Mediator Memento Observer ✓ State ✓ Strategy ✓ Visitor ✓



Overview of this Course

- We apply many GoF patterns in the context of a case study app

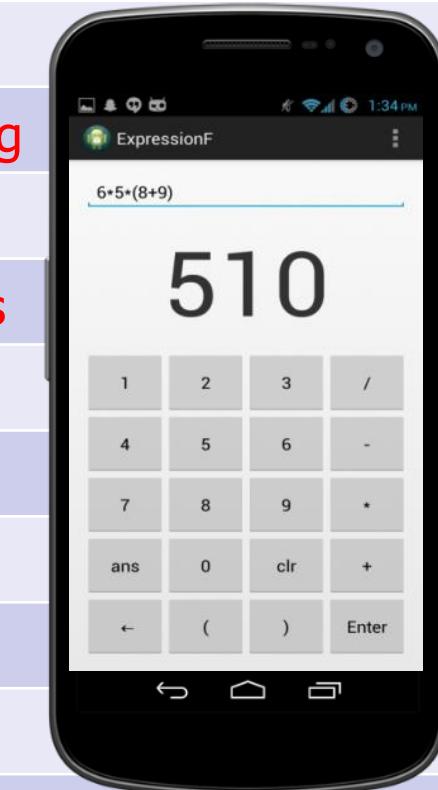
Design Problem	Pattern
Non-extensible & error-prone designs	Composite
Minimizing impact of variability	Bridge
Inflexible expression input processing	Interpreter
Inflexible interpreter output	Builder
Scattered operation implementations	Command
Inflexible creation of variabilities	Factory Method
Inflexible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



Overview of this Course

- We apply many GoF patterns in the context of a case study app (Day 1)

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



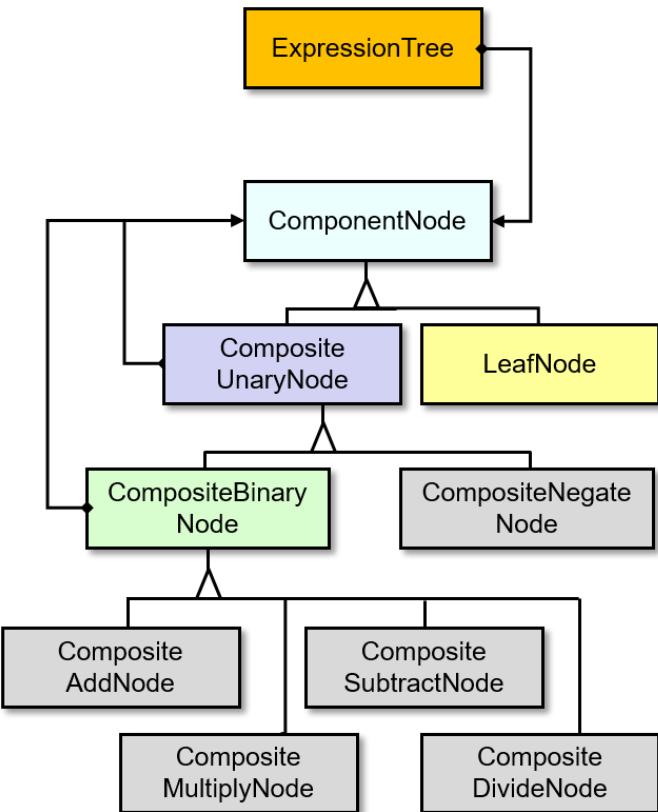
Overview of this Course

- We apply many GoF patterns in the context of a case study app (Day 2)

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

Overview of this Course

- This course focuses on both pattern-oriented design & implementation topics



```
ExpressionTree  
exprTree = ...;  
Visitor printVisitor  
= ...;  
  
for (ExpressionTree node  
: exprTree)  
node  
.accept(printVisitor);
```

Overview of this Course

- The Java expression tree processing app we cover is available on github

AndroidGUI	updates	4 months ago
CommandLine	updates	4 months ago
original	updates	4 months ago
README.md	updates	4 months ago

README.md

This folder contains source code for the following apps that implement pattern-oriented variants of ExpressionTree

original -- This folder provides the original implementation of this app from my [Design Patterns in Java LiveLesson](#) tutorial. This version compiles/runs in Eclipse configured with Android.

CommandLine -- This folder provides an updated command-line-only version of my LiveLessons tutorial that uses Java 8 features, such as lambdas and method references. This version compiles/runs in IntelliJ.

AndroidGUI -- This folder provides an updated Android GUI-only version of my LiveLessons tutorial that uses Java 8 features, such as lambdas and method references. This version compiles/runs in Android Studio.

See github.com/douglascraigschmidt/LiveLessons/tree/master/ExpressionTree

Accessing Java Features & Functionality

Accessing Java Features & Functionality

- The Java runtime environment (JRE) supports Java features

Overview Downloads Documentation Community Technologies Training

Java SE Downloads

 [DOWNLOAD ↴](#)
Java Platform (JDK) 8u101 / 8u102

 [DOWNLOAD ↴](#)
NetBeans with JDK 8

Java Platform, Standard Edition

Java SE 8u101 / 8u102
Java SE 8u101 includes important security fixes. Oracle strongly recommends that all Java SE 8 users upgrade to this release. Java SE 8u102 is a patch-set update, including all of 8u101 plus additional features (described in the release notes).
[Learn more ➔](#)

<ul style="list-style-type: none">▪ Installation Instructions▪ Release Notes▪ Oracle License▪ Java SE Products▪ Third Party Licenses▪ Certified System Configurations▪ Readme Files<ul style="list-style-type: none">▪ JDK ReadMe▪ JRE ReadMe	<p>JDK DOWNLOAD ↴</p> <p>Server JRE DOWNLOAD ↴</p> <p>JRE DOWNLOAD ↴</p>
--	--

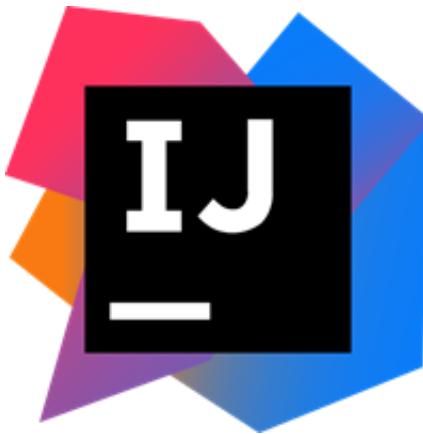


Accessing Java Features & Functionality

- The Java runtime environment (JRE) supports Java features
 - IntelliJ & Eclipse are popular Java IDEs



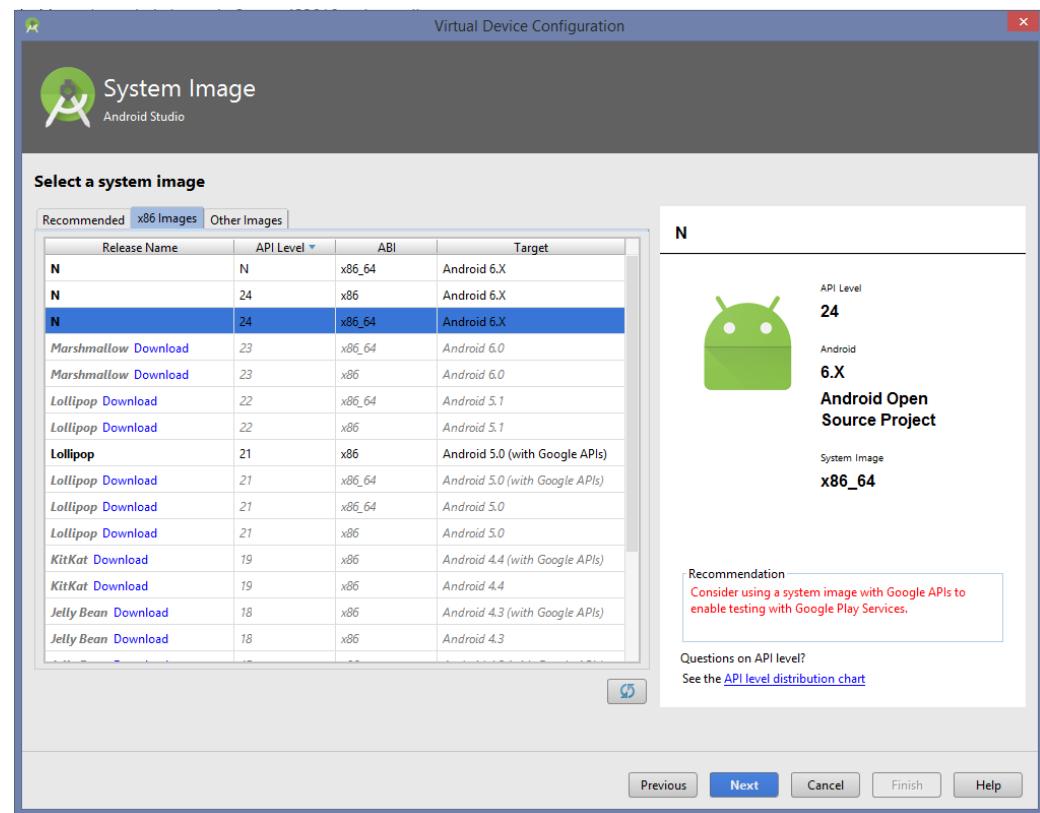
NetBeans



See www.eclipse.org/downloads & www.jetbrains.com/idea/download

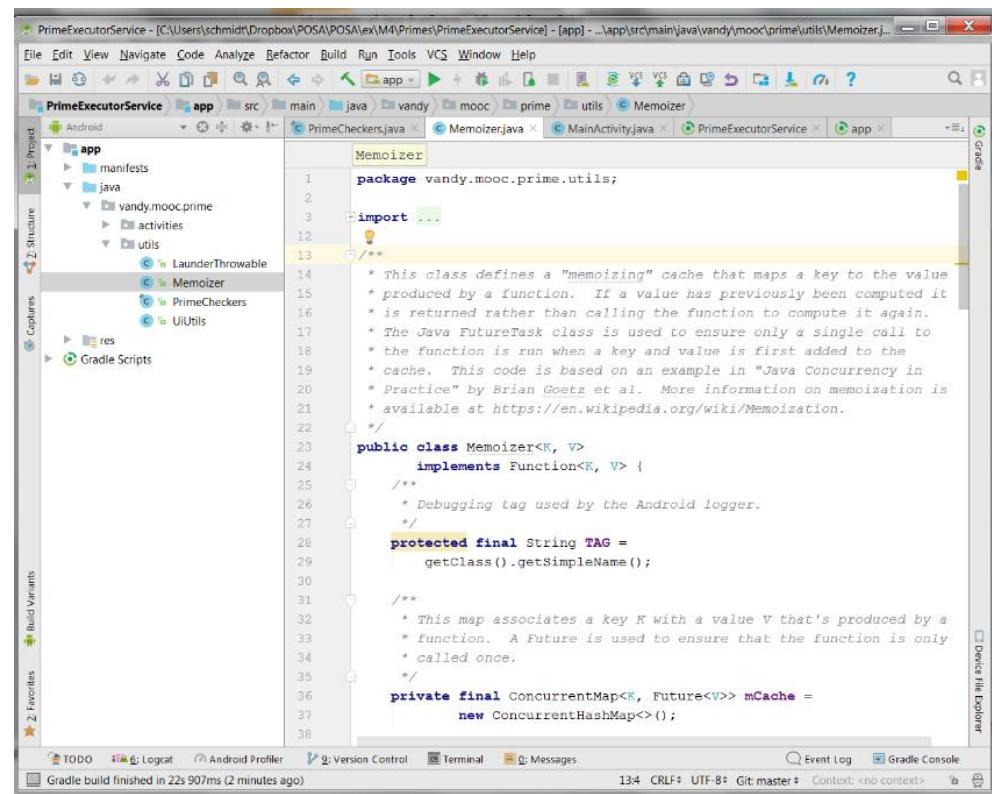
Accessing Java Features & Functionality

- The Java runtime environment (JRE) supports Java features
 - IntelliJ & Eclipse are popular Java IDEs
 - Most Java features are supported by Android API level 24 (& beyond)



Accessing Java Features & Functionality

- The Java runtime environment (JRE) supports Java features
 - IntelliJ & Eclipse are popular Java IDEs
 - Most Java features are supported by Android API level 24 (& beyond)
 - Make sure to get Android Studio 3.x or later if you use Java 8!



See developer.android.com/studio/preview/features/java8-support.html

Accessing Java Features & Functionality

- Java source code is available online
 - For browsing
grepcode.com/file/repository
grepcode.com/java/root/jdk/openjdk/8-b132/java
 - For downloading
jdk8.java.net/download.html



The screenshot shows the Java.net website with the following content:

- Java.net** logo and tagline "The Source for Java Technology Collaboration".
- Login | Register | Help** links.
- JDK 8** sidebar menu: Downloads, Feedback Forum, OpenJDK, Planet JDK.
- JDK 8 Project** title and subtitle "Building the next generation of the JDK platform".
- We Want Contributions!** callout.
- JDK 8 snapshot builds** section with a bulleted list:
 - Download 8u40 early access snapshot builds
 - Source code (instructions)
 - Official Java SE 8 Reference Implementations
 - Early Access Build Test Results (instructions)
- Feedback** section with instructions for reporting bugs.
- Helpful links** on the right: Frustrated with a bug that never got fixed? Have a great idea for improving the Java SE platform? See how to contribute for information on making contributions to the platform.

Other Digital Learning Resources

Other Digital Learning Resources

- Addition pattern topics not covered here appear in my LiveLessons course

The image shows the cover of a book titled "Design Patterns in Java LiveLessons" by Douglas C. Schmidt. The cover has a dark grey header with the word "Introduction" in white. Below the header is a large orange section containing the title "Design Patterns in Java LiveLessons" in white, sans-serif font. A large play button icon is centered on the orange background. The author's name, "Douglas C. Schmidt", is in white at the bottom left. At the bottom right is the "livelessons" logo with the tagline "video instruction from technology experts". The right side of the image shows a screenshot of the livelesson interface, featuring a video player showing a man speaking, a code editor with Java code, and a terminal window displaying command-line output.

Introduction

Design Patterns in Java LiveLessons

Douglas C. Schmidt

livelessons
video instruction from technology experts

©2014 Pearson, Inc.

See www.dre.vanderbilt.edu/~schmidt/LiveLessons/DPiJava

Other Digital Learning Resources

- There's a Facebook group dedicated to discussing Java design pattern topics



See www.facebook.com/groups/623398741056491

Other Digital Learning Resources

- There are several related Live Training courses are coming soon on Java 8

Programming with Java 8 Lambdas and Streams



October 16th, 2018
9:00am – 12:00pm CST

[SEE PRICING OPTIONS](#)

109 spots available
Registration closes February 28, 2018 5:00 PM

Scalable Programming with Java 8 Parallel Streams



August 20th, 2018
10:00am – 2:00pm CST

[SEE PRICING OPTIONS](#)

40 spots available
Registration closes January 9, 2018 5:00 PM

Reactive Programming with Java 8 CompletableFuture



DOUGLAS SCHMIDT



October 4th, 2018
9:00am – 1:00pm CST

[SEE PRICING OPTIONS](#)

20 spots available
Registration closes January 11, 2018 5:00 PM

Scalable Concurrency with the Java Executor Framework



DOUGLAS SCHMIDT



October 29, 2018
11:00am – 2:00pm CDT

[SEE PRICING OPTIONS](#)

194 spots available
Registration closes October 28, 2018 5:00 PM

See www.dre.vanderbilt.edu/~schmidt/DigitalLearning

Other Digital Learning Resources

- See my website for many more videos & screencasts related to programming with patterns, frameworks, Java, etc.



Digital Learning Offerings

Douglas C. Schmidt (d.schmidt@vanderbilt.edu)
Associate Chair of Computer Science and Engineering,
Professor of Computer Science, and Senior Researcher
in the Institute for Software Integrated Systems (ISIS)
at Vanderbilt University



O'Reilly LiveTraining Courses

- Programming with Java 8 Lambdas and Streams
 - [January 9th, 2018, 9:00am-12:00pm central time](#)
 - [February 1st, 2018, 9:00am-12:00pm central time](#)
 - March 1st, 2018, 9:00am-12:00pm central time
- Scalable Programming with Java 8 Parallel Streams
 - [January 10th, 2018, 11:00am-3:00pm central time](#)
 - February 6th, 2018, 11:00am-3:00pm central time
 - March 6th, 2018, 11:00am-3:00pm central time
- Reactive Programming with Java 8 Completable Futures
 - [January 12th, 2018, 10:00am-1:00pm central time](#)
 - [February 13th, 2018, 10:00am-2:00pm central time](#)
 - March 13th, 2018, 10:00am-2:00pm central time

Pearson LiveLessons Courses

- [Java Concurrency](#)
- [Design Patterns in Java](#)

Coursera MOOCs

- [Android App Development](#) Coursera Specialization
- [Pattern-Oriented Software Architecture](#) (POSA)

Vanderbilt University Courses

- [Playlist](#) from my [YouTube Channel](#) videos from [CS 891: Introduction to Concurrent and Parallel Java Programming with Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 892: Concurrent Java Programming with Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 251: Intermediate Software Design with Java](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 282: Concurrent Java Network Programming in Android](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 251: Intermediate Software Design with C++](#)
- [Playlist](#) from my [YouTube Channel](#) videos from [CS 282: Systems Programming for Android](#)

See www.dre.vanderbilt.edu/~schmidt/DigitalLearning

Other Digital Learning Resources

- See my website for many more videos & screencasts related to programming with patterns, frameworks, Java, etc.
 - Videos from my MOOC “Pattern-Oriented Software Architecture” are relevant!

The image shows a YouTube playlist page. At the top, there's a video thumbnail for 'Section 2: Introduction to Patterns and Frameworks' by Douglas Schmidt, which is 11:20 long. Below it is another thumbnail for 'Overview of Patterns (Part 1)' by Douglas Schmidt, 14:14 long. The main title of the playlist is 'Pattern-Oriented Software Architectures for Concurrent and Networked Software'. It says there are 81 videos and 2,146 views, last updated on Jan 4, 2018. A bio for Douglas Schmidt is present, along with a note about the videos being from 2013. The bottom part of the image shows a list of 10 more video thumbnails, each with a title, duration, and author.

Video Number	Title	Duration	Author
17	Section 2: Introduction to Patterns and Frameworks	11:20	Douglas Schmidt
18	Overview of Patterns (Part 1)	14:14	Douglas Schmidt
19	Overview of Patterns (Part 2)	22:59	Douglas Schmidt
20	Overview of Patterns (Part 3)	25:03	Douglas Schmidt
21	GoF and POSA Pattern Examples (Part 1)	11:46	Douglas Schmidt
22	GoF and POSA Pattern Examples (Part 2)	21:34	Douglas Schmidt
23	GoF and POSA Pattern Examples (Part 3)	21:57	Douglas Schmidt

See www.youtube.com/playlist?list=PLZ9NgFYEMxp6CHE-QQ040tIDILNcBqJnc

Other Digital Learning Resources

- See my website for many more videos & screencasts related to programming with patterns, frameworks, Java, etc.
 - Videos from my MOOC “Pattern-Oriented Software Architecture” are relevant!



YouTube Search

Section 2: Introduction to Patterns and Frameworks Douglas Schmidt 11:20

Overview of Patterns (Part 1) Douglas Schmidt 14:14

Overview of Patterns (Part 2) Douglas Schmidt 22:59

Overview of Patterns (Part 3) Douglas Schmidt 25:03

GoF and POSA Pattern Examples (Part 1) Douglas Schmidt 11:46

GoF and POSA Pattern Examples (Part 2) Douglas Schmidt 21:34

GoF and POSA Pattern Examples (Part 3) Douglas Schmidt 21:57

Pattern-Oriented Software Architectures for Concurrent and Networked Software

81 videos • 2,146 views • Last updated on Jan 4, 2018

Douglas Schmidt EDIT

These videos were filmed in 2013 as part of my original Coursera MOOC on pattern-oriented software architectures for concurrent and networked software. The course is organized as follows:

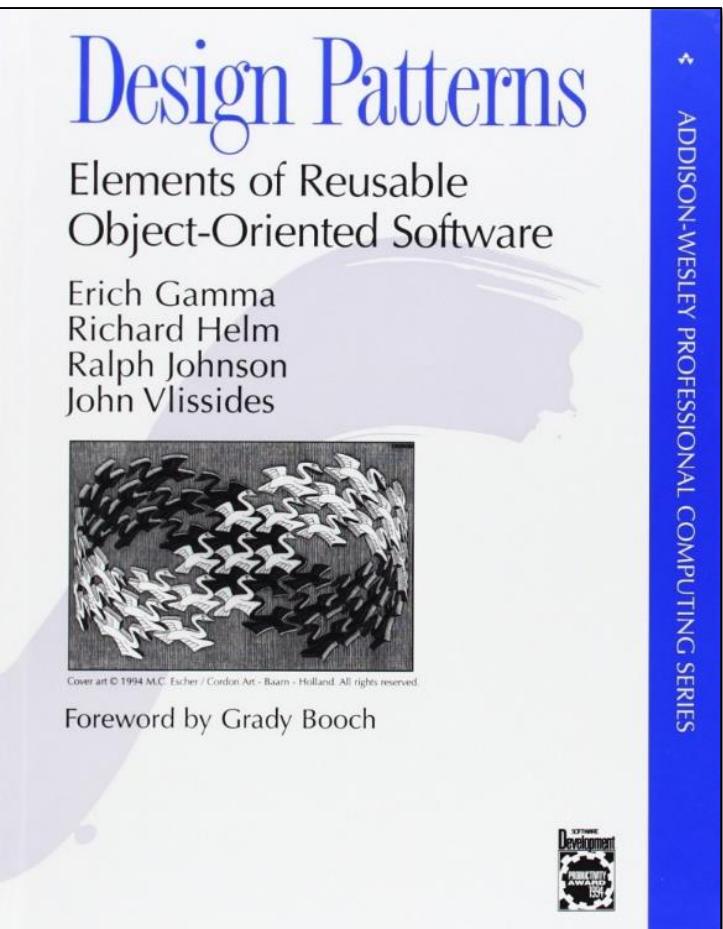
* Section 0: Overview of all the topics covered in this course. This material is designed to help you visualize the motivations for—and challenges of—concurrent and networked software. I also summarize how

The image shows a YouTube channel page for a course titled "Pattern-Oriented Software Architectures for Concurrent and Networked Software". The channel has 2,146 views and was last updated on January 4, 2018. It contains 81 videos. The channel description explains that the videos were filmed in 2013 as part of a Coursera MOOC and provides an overview of the course organization, mentioning Section 0 and the GoF and POSA pattern examples.

We'll cover some of these topics in this course as time permits

Other Digital Learning Resources

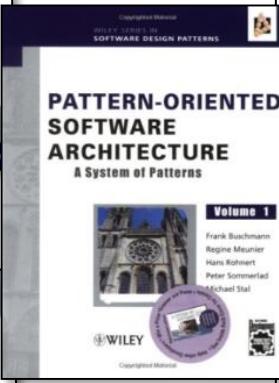
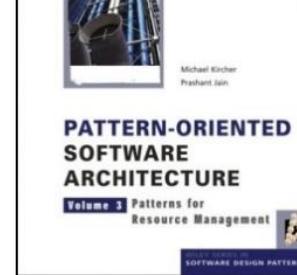
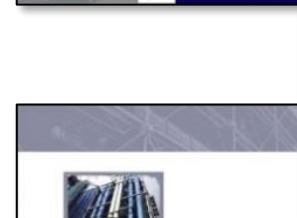
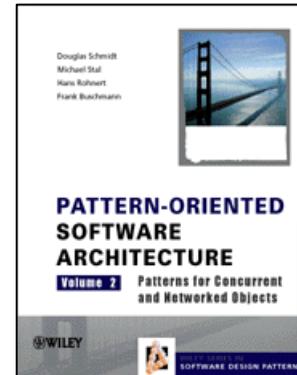
- The book *Design Patterns: Elements of Reusable Object-Oriented Software* describes all the Gang-of-Four (GoF) patterns in detail



See en.wikipedia.org/wiki/Design_Patterns

Other Digital Learning Resources

- The “POSA” books contain good sources of material on other types of patterns & pattern relationships



See www.dre.vanderbilt.edu/~schmidt/POSA

End of Course Overview

Overview of the Expression Tree

Processing App Case Study

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

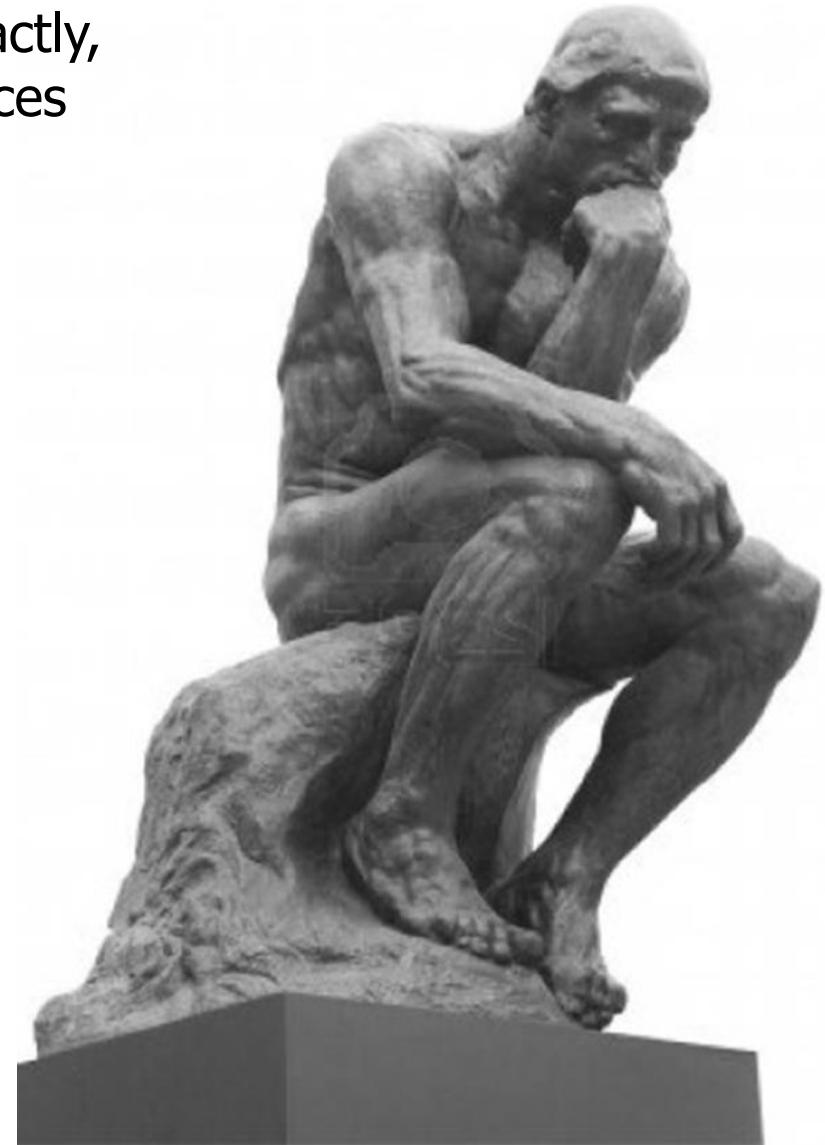
**Institute for Software
Integrated Systems**

**Vanderbilt University
Nashville, Tennessee, USA**



Lesson Intro

- While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities



"Sitting & thinking" is not sufficient...

Lesson Intro

- While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities
- Instead, it's usually better to see how patterns can help improve non-trivial programs



Lesson Intro

- While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities
- Instead, it's usually better to see how patterns can help improve non-trivial programs, e.g.
 - Easier to write & read
 - Easier to maintain & modify
 - More efficient & robust



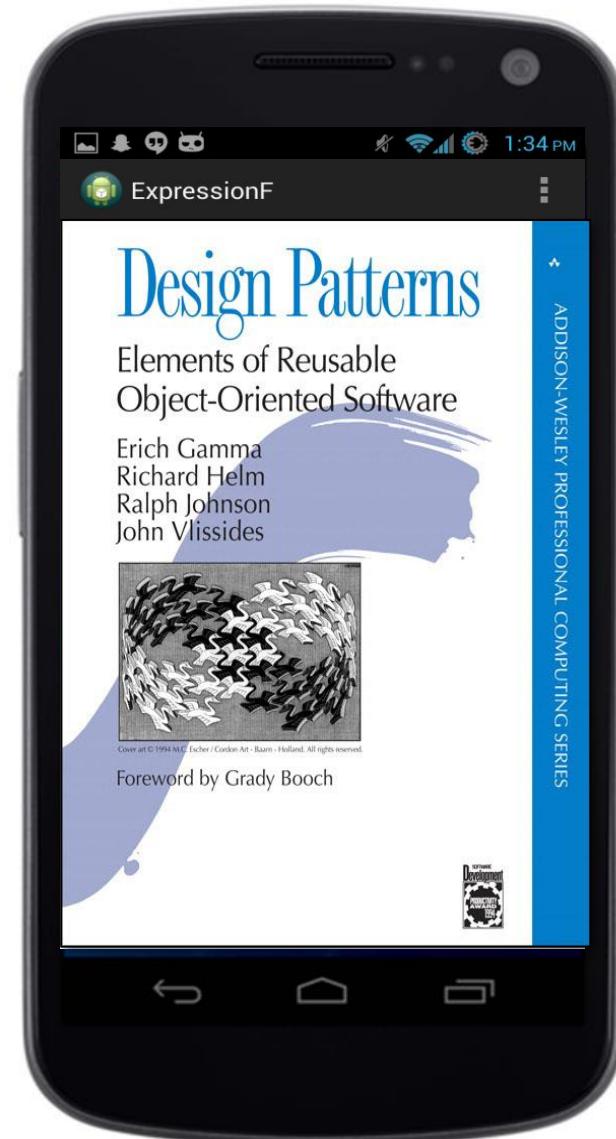
Lesson Intro

- While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities
- Instead, it's usually better to see how patterns can help improve non-trivial programs
- This lesson describes a realistic—yet tractable—expression tree processing app we'll use as a case study throughout the course



Lesson Intro

- While patterns can be discussed abstractly, effective design & programming practices are not learned best by generalities
- Instead, it's usually better to see how patterns can help improve non-trivial programs
- This lesson describes a realistic—yet tractable—expression tree processing app we'll use as a case study throughout the course
 - This case study applies many “Gang of Four” (GoF) patterns



See en.wikipedia.org/wiki/Design_Patterns

Learning Objectives

Learning Objectives

- Understand the goals of the object-oriented (OO) expression tree case study

```
Console X
Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Jul 15, 2013 12:51:24 PM)

1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order| level-order]
0. quit

> format in-order

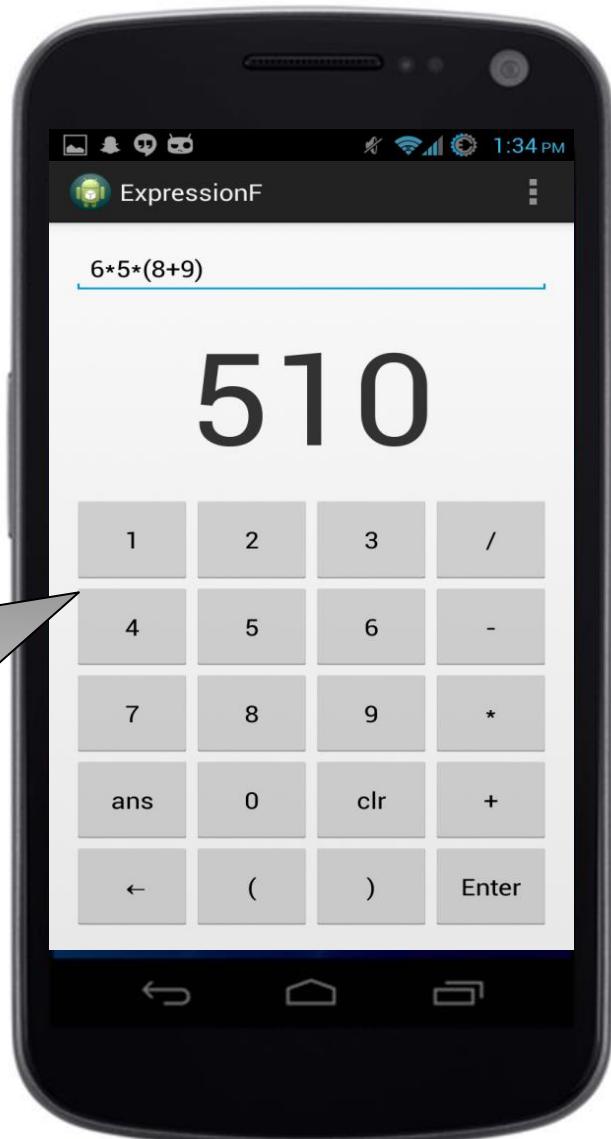
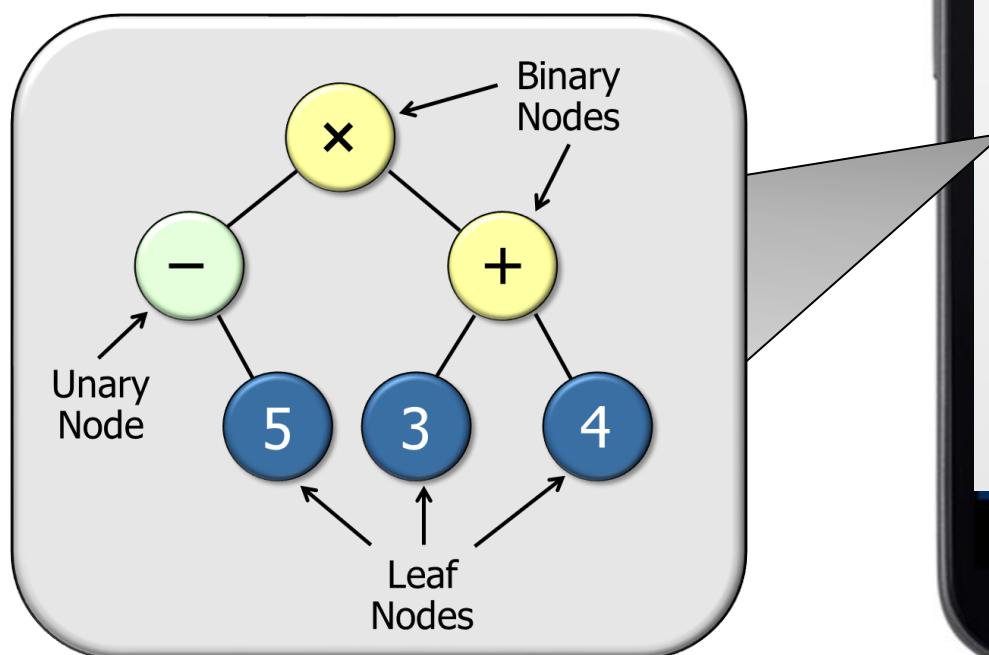
1. expr [expression]
2a. eval [post-order]
2b. print [in-order | pre-order | post-order| level-order]
0b. set [post-order]
0c. quit

> expr 6*5*(8+9)
```



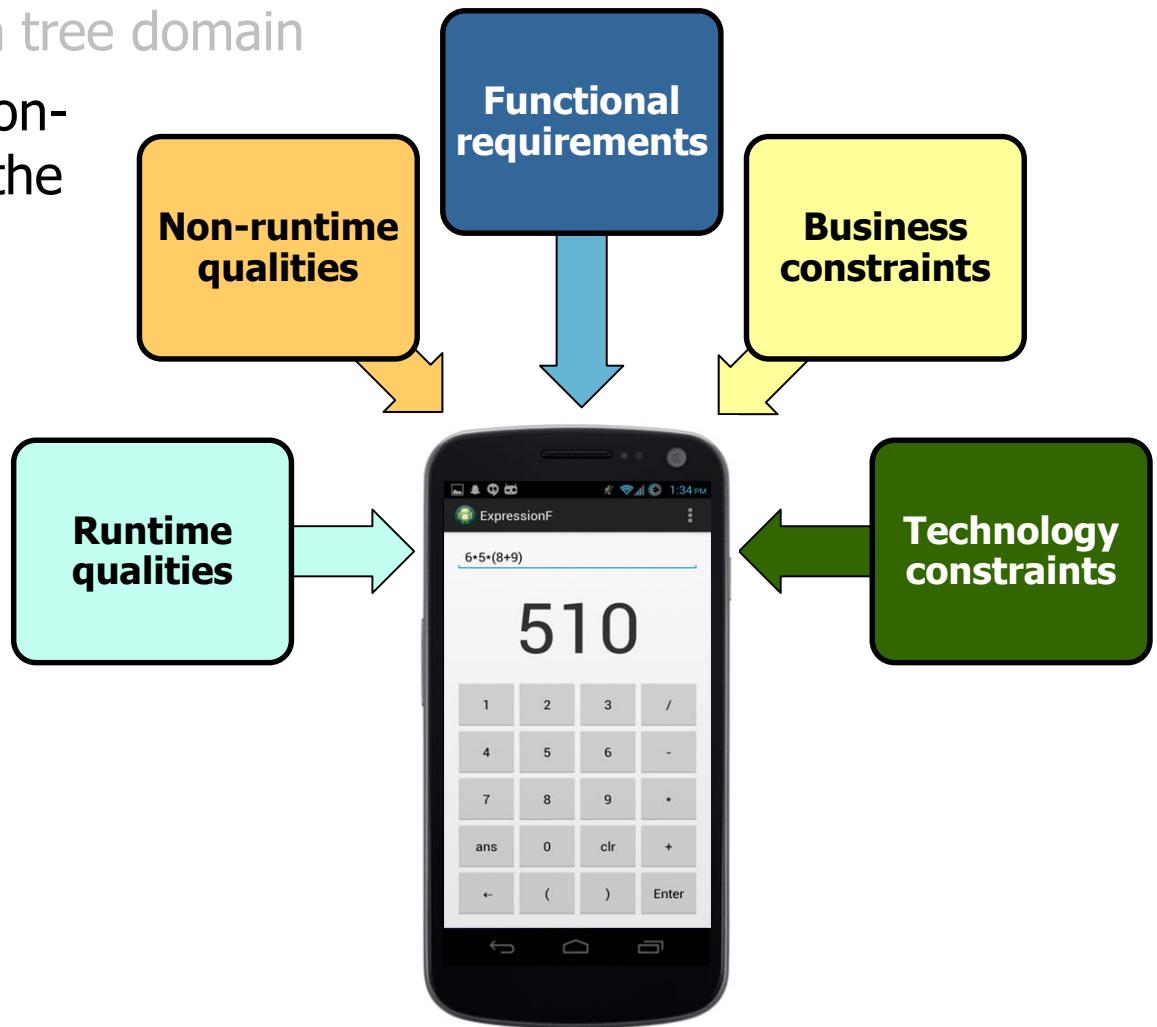
Learning Objectives

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



Learning Objectives

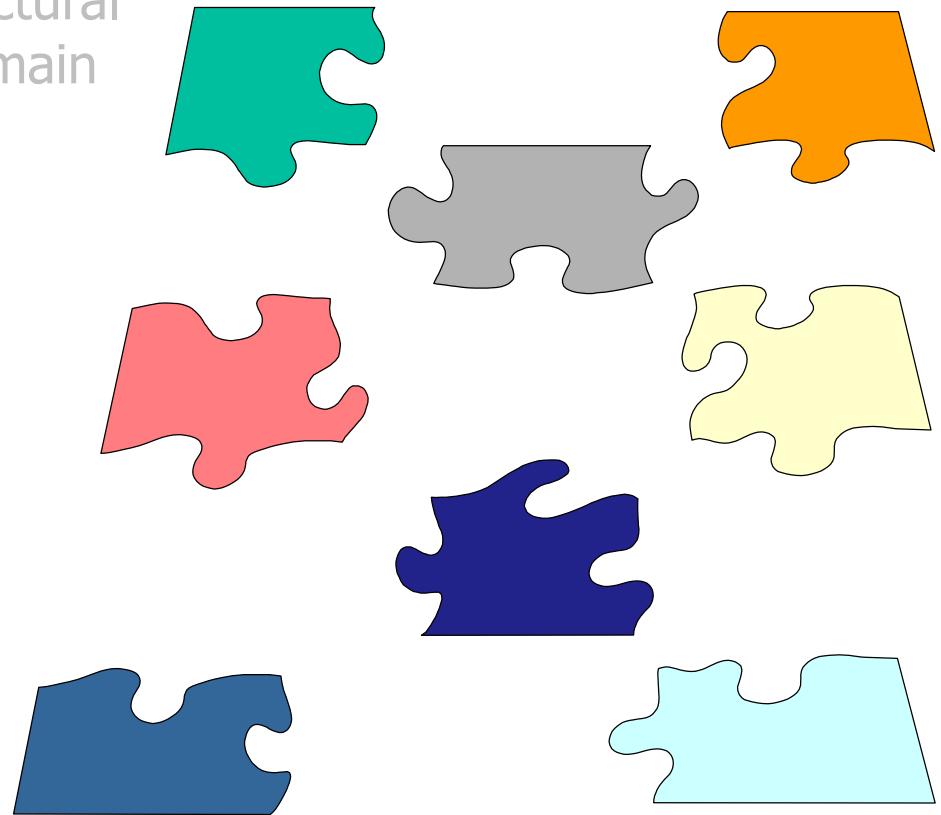
- Understand the goals of the object-oriented (OO) expression tree case study
- Recognize the key behavioral & structural properties in the expression tree domain
- Evaluate the functional & non-functional requirements of the case study



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

Learning Objectives

- 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



Expression Tree Processing

App Case Study Goals

Expression Tree Processing App Case Study Goals

- Develop an OO expression tree processing app using *patterns & frameworks*

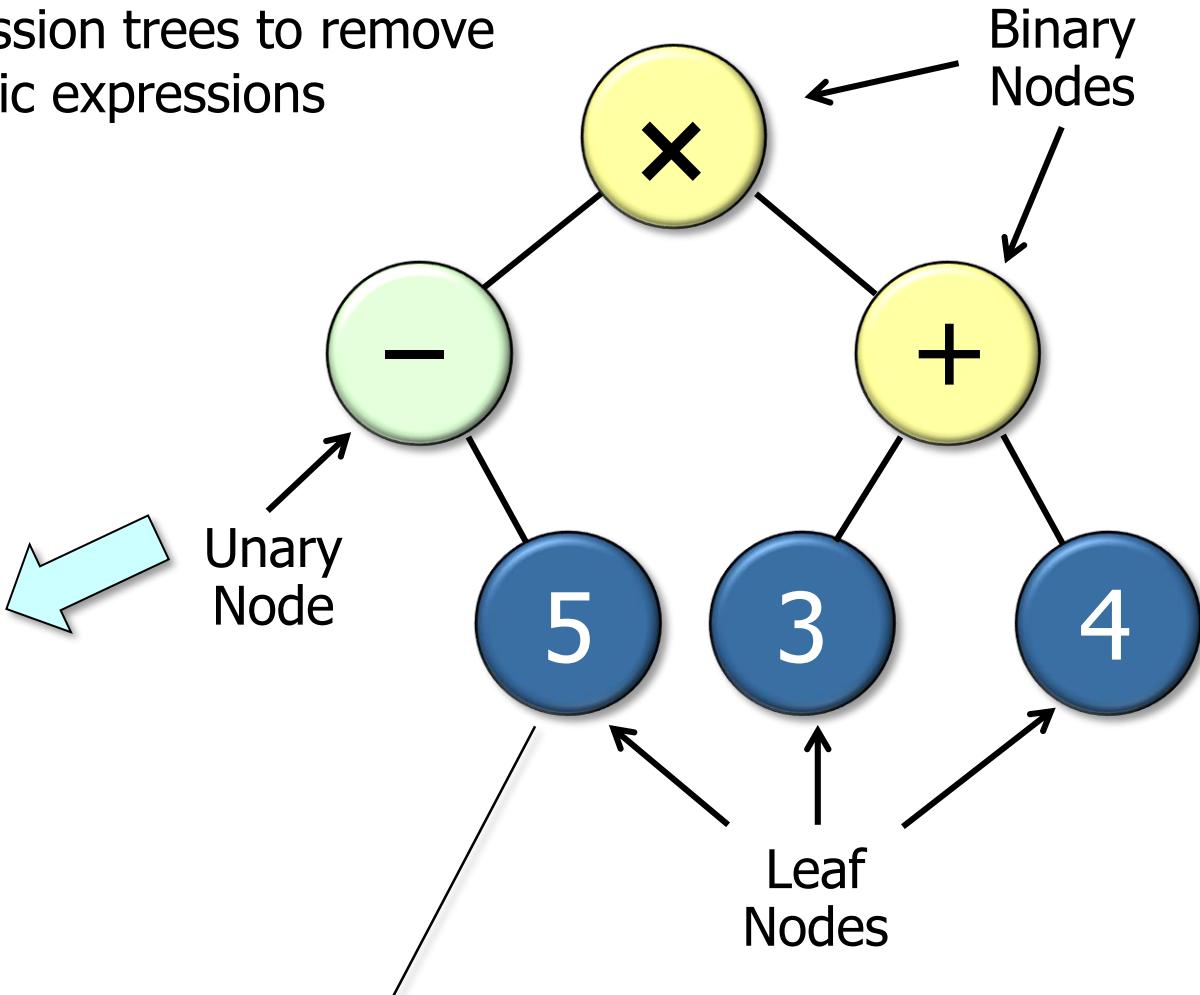


Design Problem	Pattern
Non-extensible & error-prone designs	Composite
Minimizing impact of variability	Bridge
Inflexible expression input processing	Interpreter
Inflexible interpreter output	Builder
Scattered operation implementations	Command
Inflexible creation of variabilities	Factory Method
Inflexible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

Naturally, these patterns apply to more than expression tree processing apps!

Expression Tree Processing App Case Study Goals

- This app uses expression trees to remove ambiguity in algebraic expressions

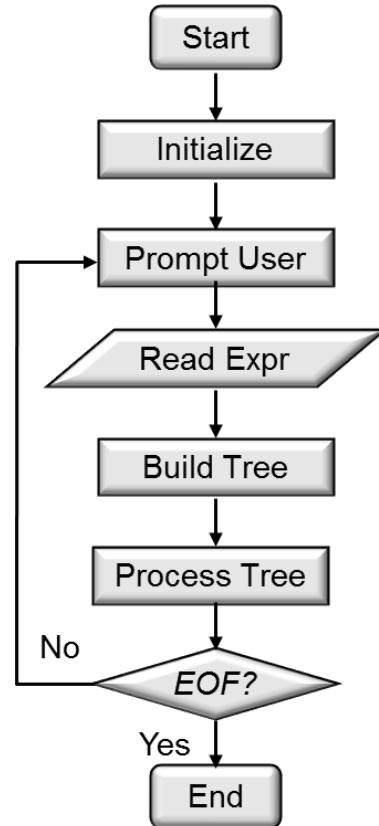
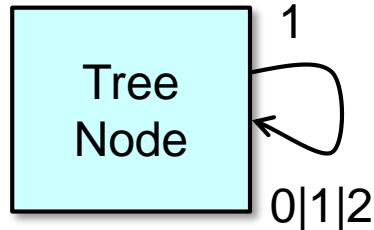


Each node of a binary expression tree has zero, one, or two children

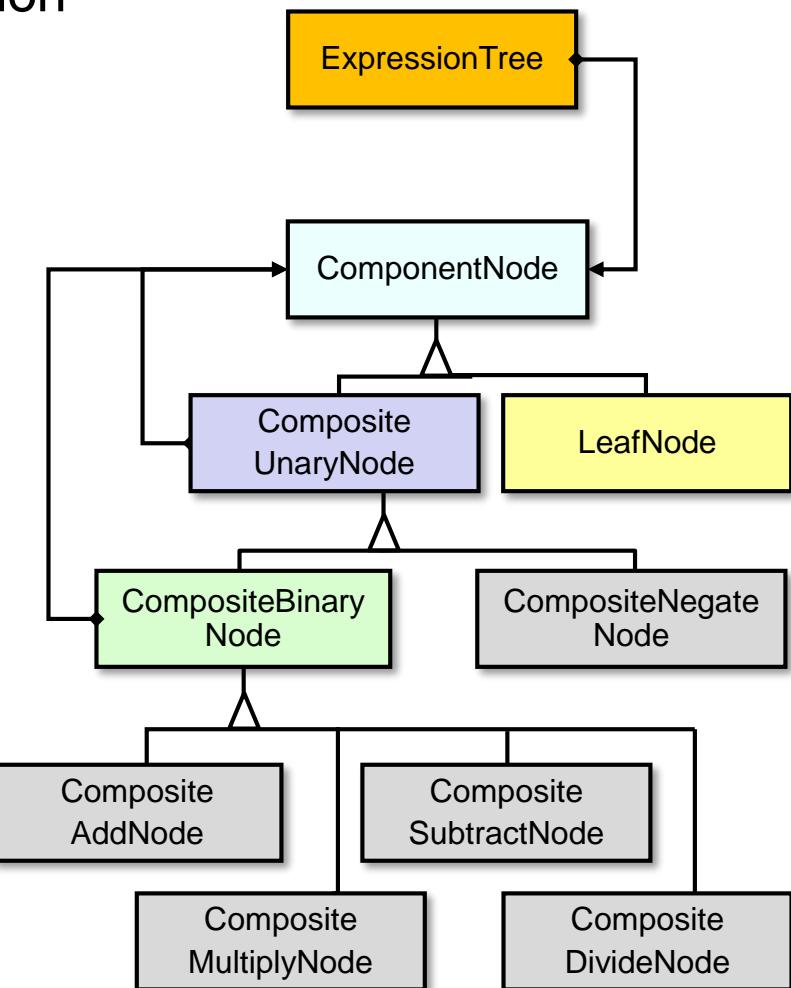
Expression Tree Processing App Case Study Goals

- Compare/contrast algorithmic decomposition & object-oriented (OO) approaches

Despite decades of OO focus, algorithmic decomposition is still surprisingly common



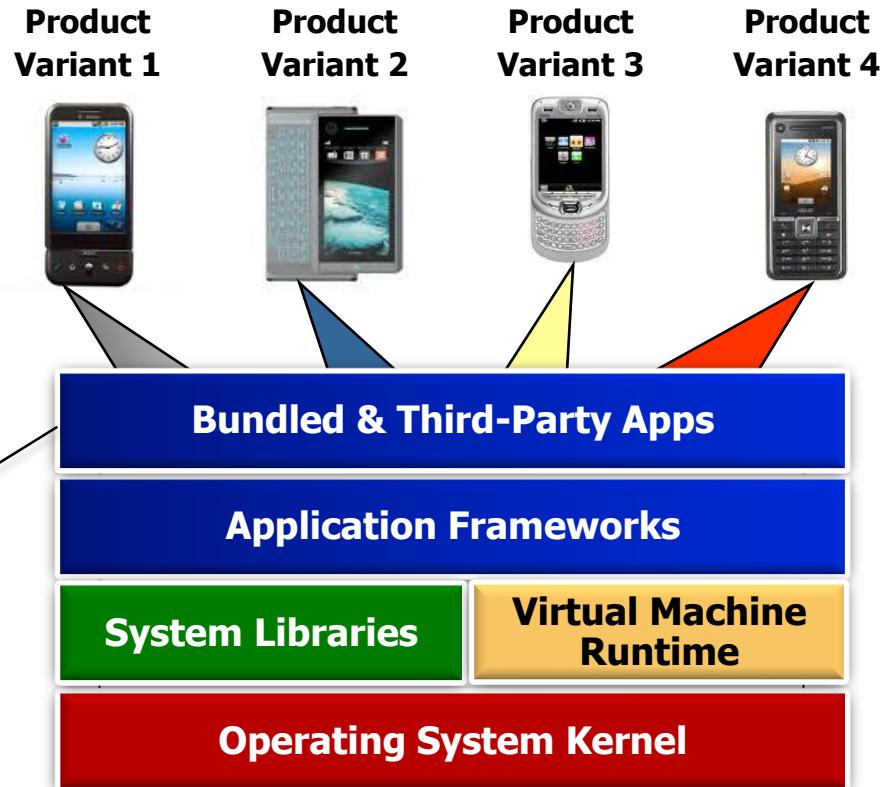
Algorithmic Decomposition



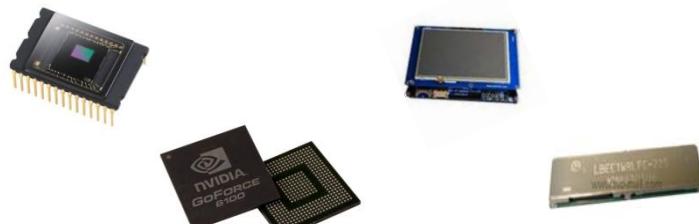
Pattern- & OO Decomposition

Expression Tree Processing App Case Study Goals

- Demonstrate *Scope, Commonality, & Variability* (SCV) analysis as a means to achieve systematic software reuse

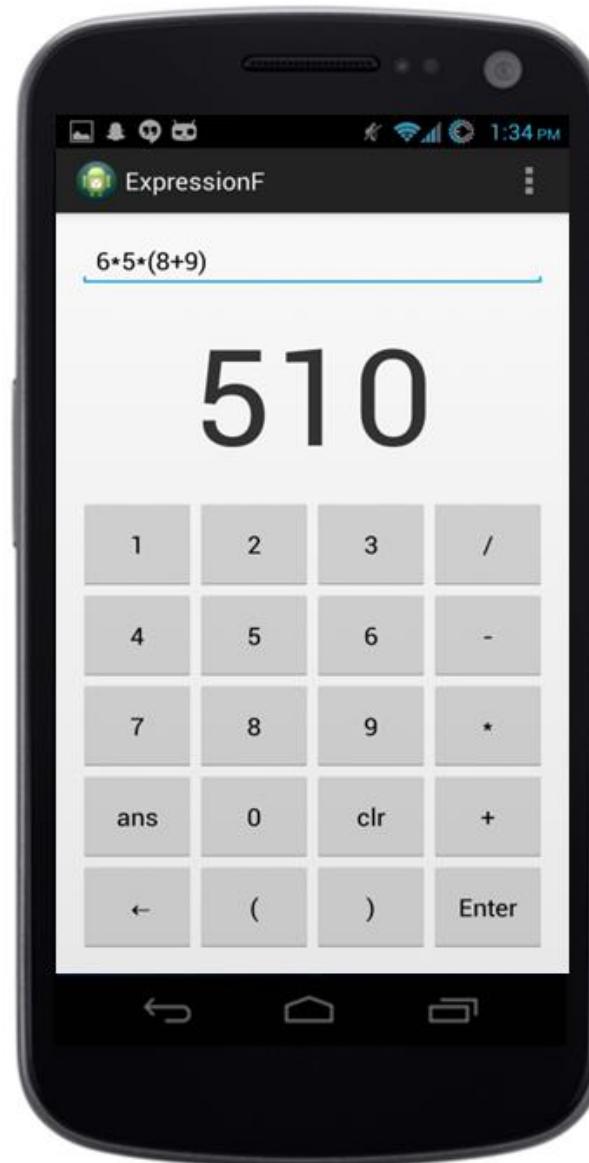
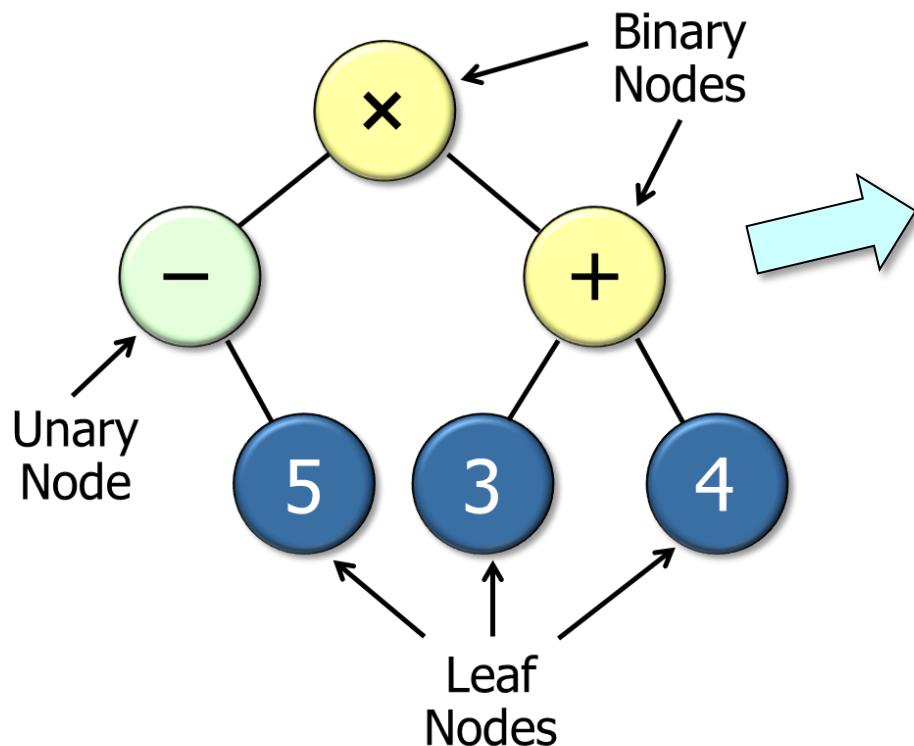


- Identify common elements of a domain & define stable interfaces*
- Identify variable elements of a domain & define stable interfaces*



Expression Tree Processing App Case Study Goals

- Demonstrate *Scope, Commonality, & Variability* (SCV) analysis as a means to achieve systematic software reuse
 - Apply SCV in the context of the expression tree processing app



Expression Tree Processing App Case Study Goals

- Show how to implement pattern-oriented OO frameworks & functional programs in Java

```
ExpressionTree exprTree = ...;  
Visitor printVisitor = ...;  
  
for (Iterator<ExpressionTree> iter  
      = exprTree.iterator();  
      iter.hasNext();  
      )  
    iter.next().accept  
      (printVisitor);  
  
  
for (ExpressionTree node : exprTree)  
  node.accept(printVisitor);  
  
  
exprTree  
  .forEach  
  (node ->  
    node.accept(printVisitor));
```

Expression Tree Processing App Case Study Goals

- Show how to implement pattern-oriented OO frameworks & functional programs in Java

```
ExpressionTree exprTree = ...;  
Visitor printVisitor = ...;  
  
for (Iterator<ExpressionTree> iter  
      = exprTree.iterator();  
      iter.hasNext();  
      )  
    iter.next().accept  
      (printVisitor);
```

Java-style Iterator pattern



```
for (ExpressionTree node : exprTree)  
  node.accept(printVisitor);
```

```
exprTree  
  .forEach  
  (node ->  
    node.accept(printVisitor));
```

Expression Tree Processing App Case Study Goals

- Show how to implement pattern-oriented OO frameworks & functional programs in Java

```
ExpressionTree exprTree = ...;  
Visitor printVisitor = ...;  
  
for (Iterator<ExpressionTree> iter  
      = exprTree.iterator();  
      iter.hasNext();  
      )  
    iter.next().accept  
      (printVisitor);
```

Java for-each loop (assumes
ExpressionTree implements
Iterable)



```
for (ExpressionTree node : exprTree)  
  node.accept(printVisitor);  
  
exprTree  
  .forEach  
  (node ->  
    node.accept(printVisitor));
```

Expression Tree Processing App Case Study Goals

- Show how to implement pattern-oriented OO frameworks & functional programs in Java

```
ExpressionTree exprTree = ...;  
Visitor printVisitor = ...;  
  
for (Iterator<ExpressionTree> iter  
      = exprTree.iterator();  
      iter.hasNext();  
      )  
    iter.next().accept  
      (printVisitor);
```

```
for (ExpressionTree node : exprTree)  
  node.accept(printVisitor);
```

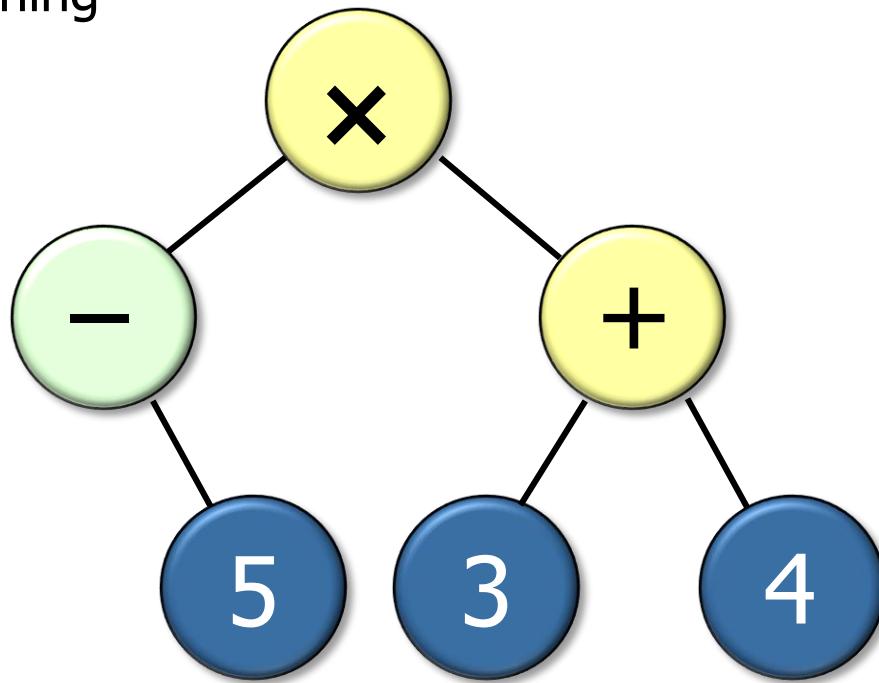
Java `forEach()` method (also assumes `ExpressionTree` implements `Iterable`)



```
exprTree  
.forEach  
(node ->  
  node.accept(printVisitor));
```

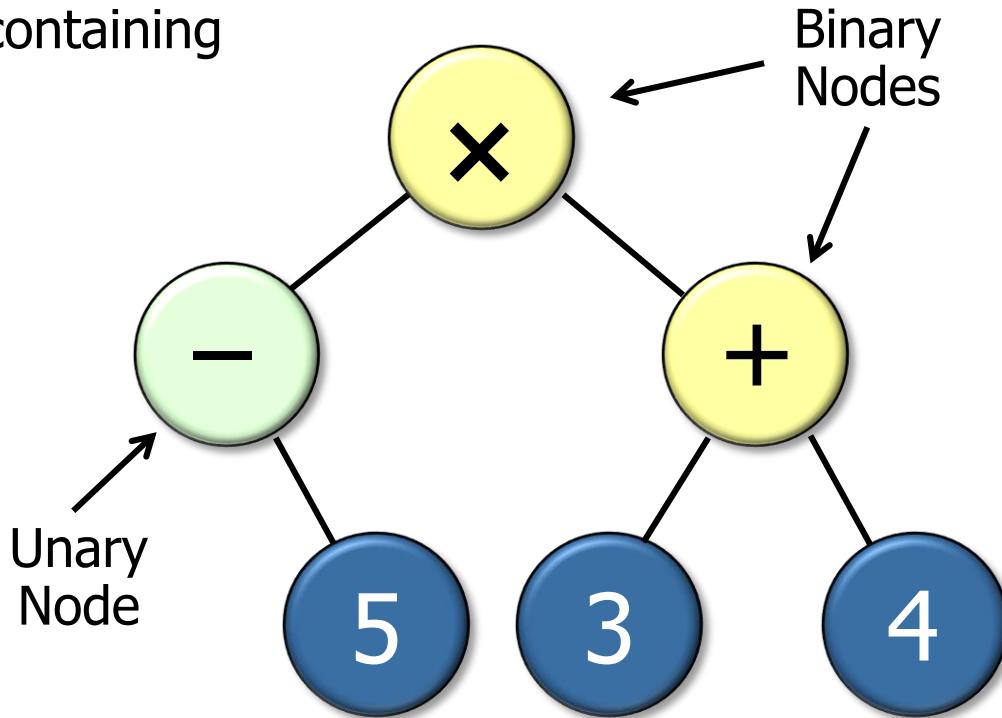
Overview of Expression Tree Processing Domain

- Expression trees consist of nodes containing *operators & operands*



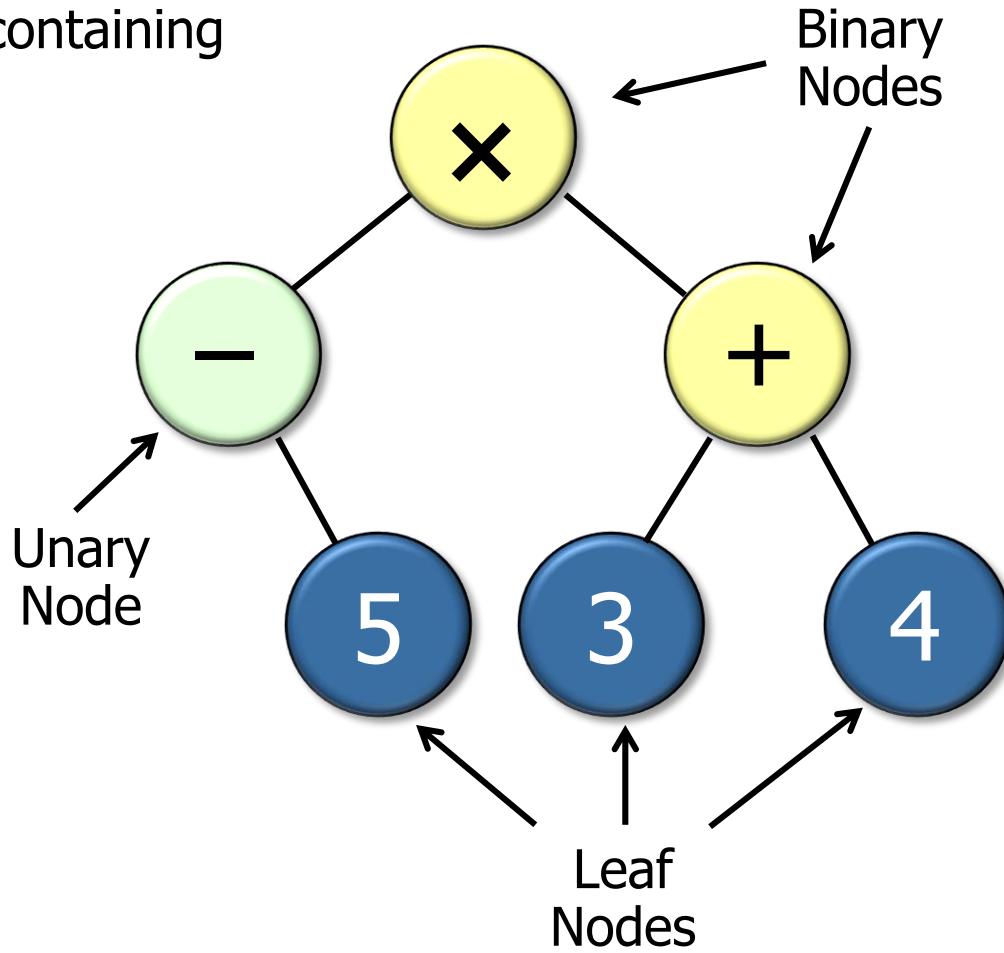
Overview of Expression Tree Processing Domain

- Expression trees consist of nodes containing *operators & operands*
 - Operators are *interior nodes* in the tree
 - i.e., *binary & unary nodes*



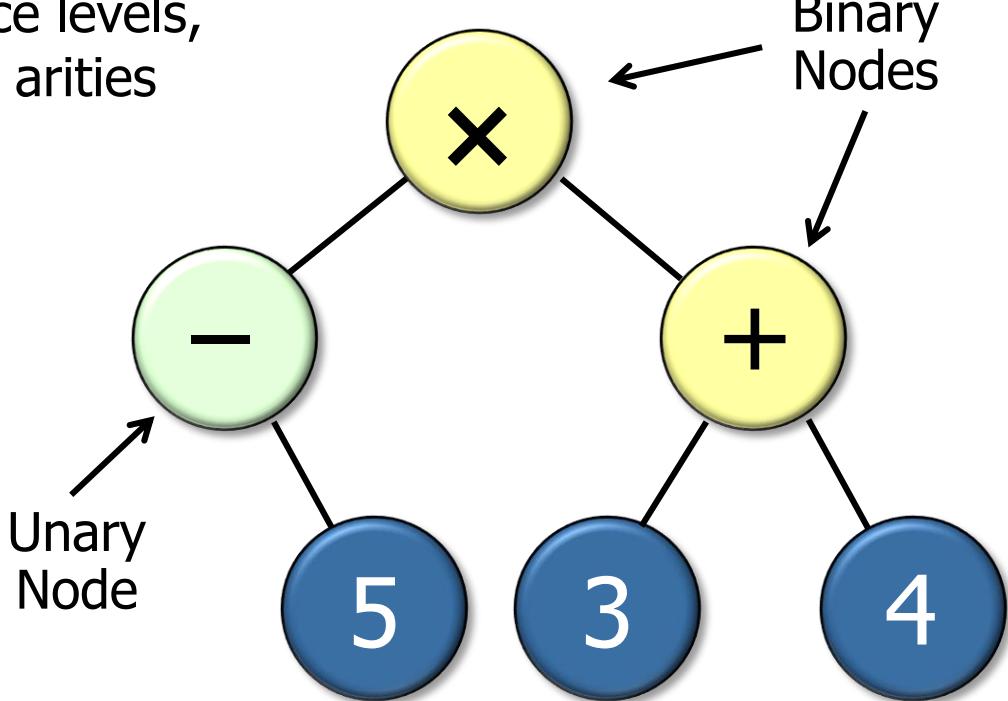
Overview of Expression Tree Processing Domain

- Expression trees consist of nodes containing *operators & operands*
 - Operators are *interior nodes* in the tree
 - Operands are *exterior nodes* in the tree
 - i.e., *leaf nodes*



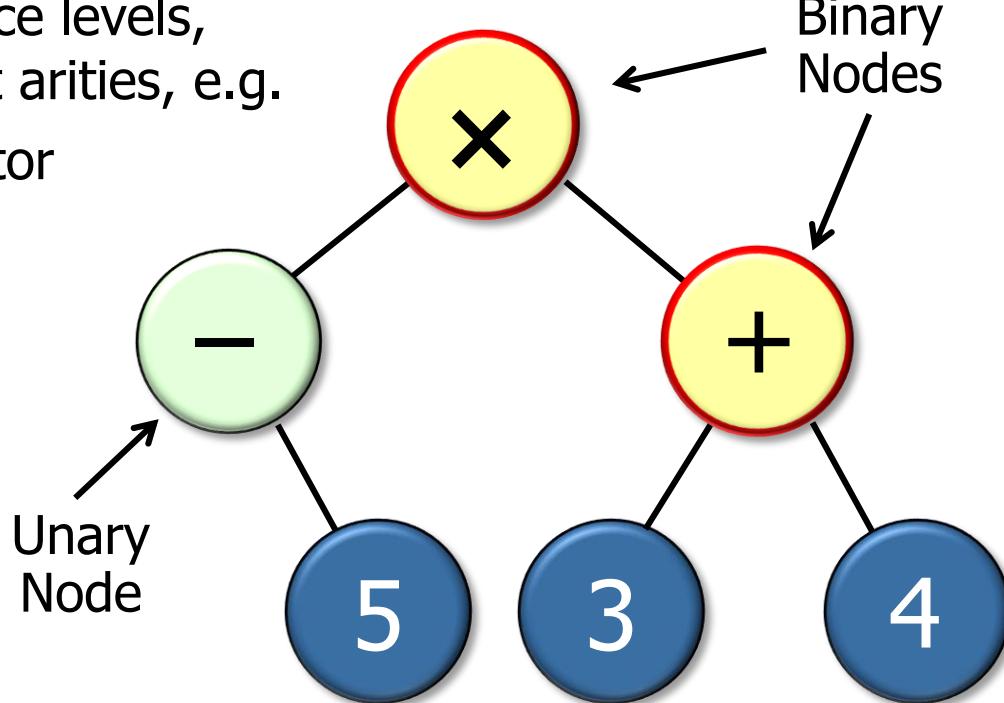
Overview of Expression Tree Processing Domain

- Operators have different precedence levels, different associativities, & different arities



Overview of Expression Tree Processing Domain

- Operators have different precedence levels, different associativities, & different arities, e.g.
 - Precedence defines which operator to perform first to evaluate a mathematical expression
 - Multiplication takes precedence over addition



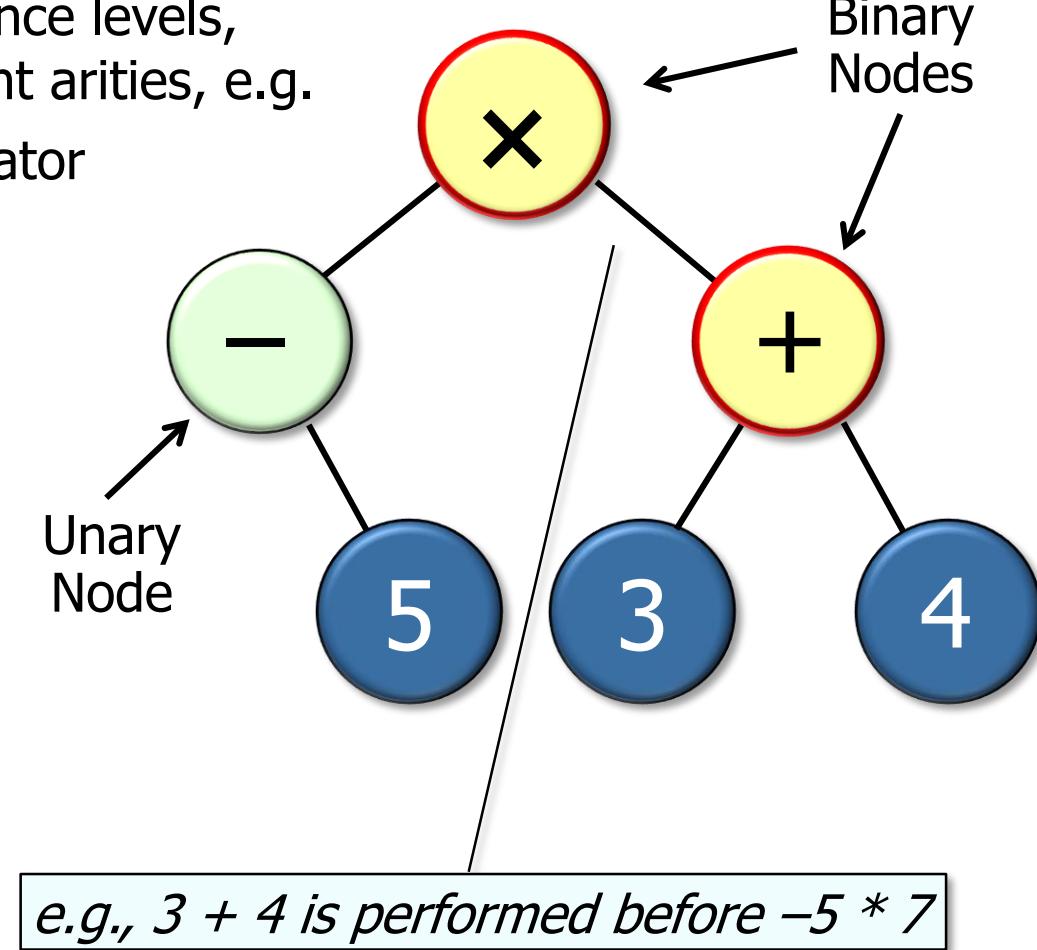
Overview of Expression Tree Processing Domain

- Operators have different precedence levels, different associativities, & different arities, e.g.

- Precedence defines which operator to perform first to evaluate a mathematical expression

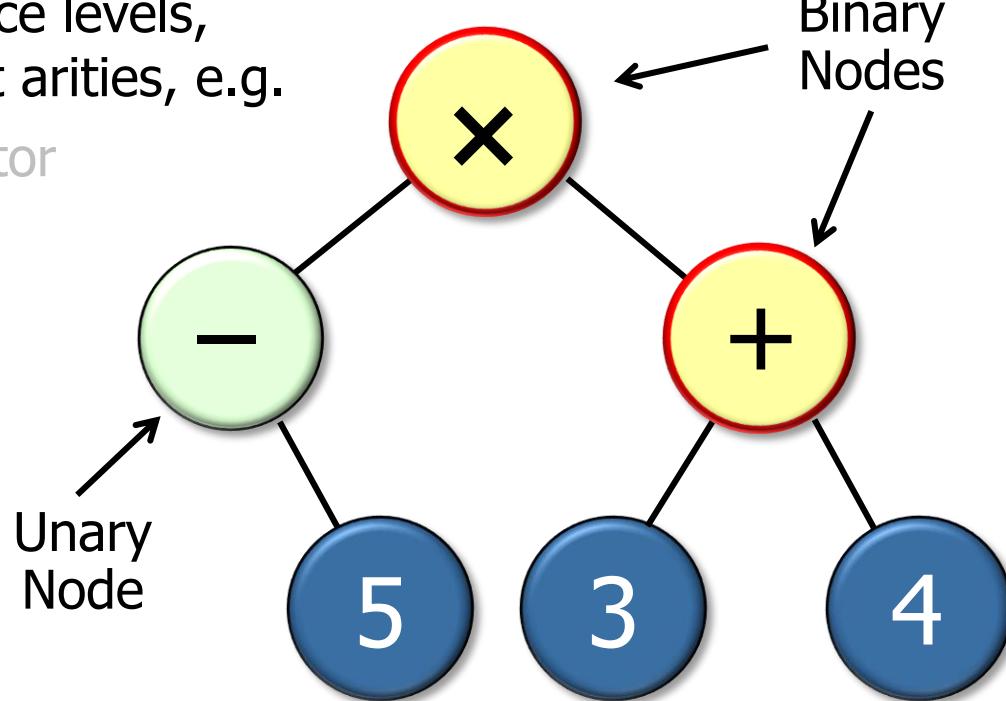
- Multiplication takes precedence over addition

- Operator locations in a tree unambiguously designate precedence



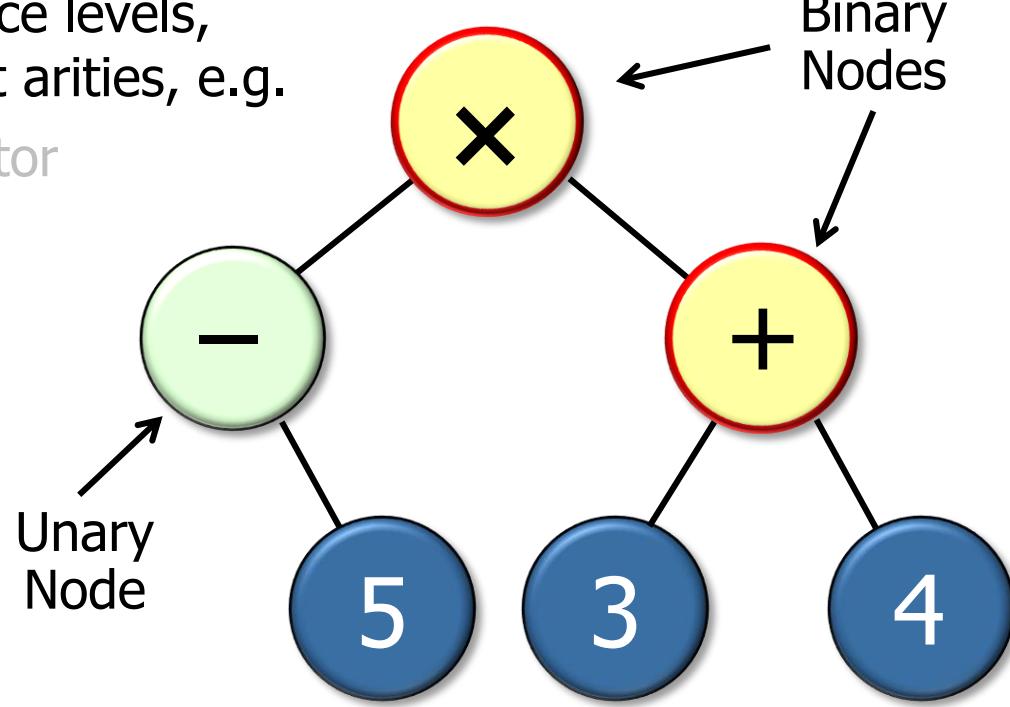
Overview of Expression Tree Processing Domain

- Operators have different precedence levels, different associativities, & different arities, e.g.
 - Precedence defines which operator to perform first to evaluate a mathematical expression
 - Associativity determines how operators of the same level of precedence are grouped in the absence of parentheses
 - $5 + 3 - 4 == (5 + 3) - 4$



Overview of Expression Tree Processing Domain

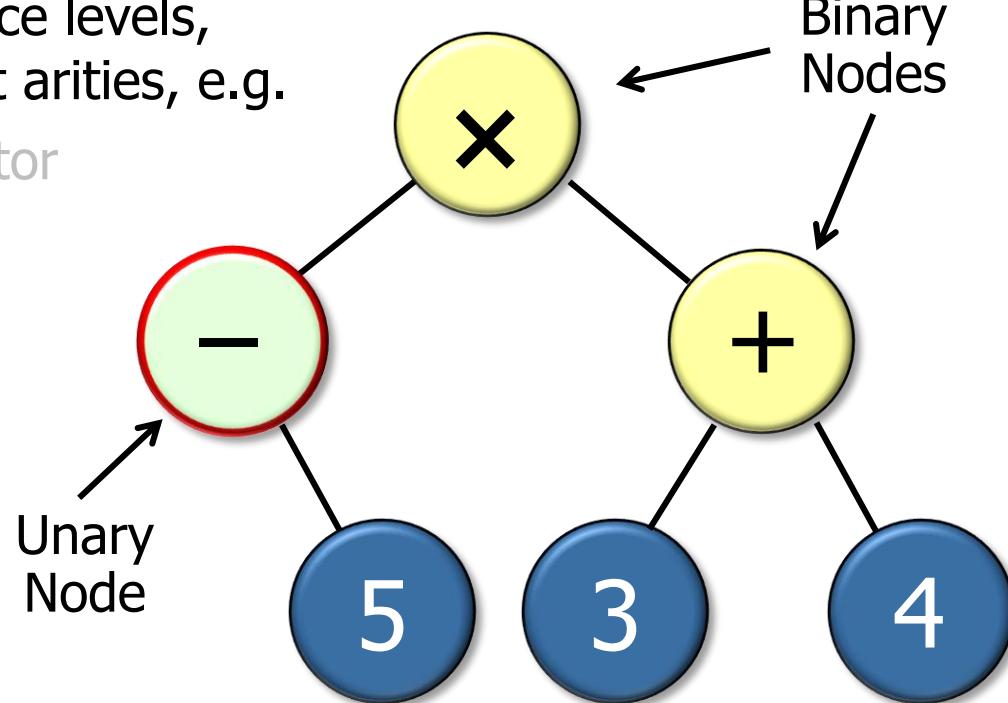
- Operators have different precedence levels, different associativities, & different arities, e.g.
 - Precedence defines which operator to perform first to evaluate a mathematical expression
 - Associativity determines how operators of the same level of precedence are grouped in the absence of parentheses
 - Arity defines the # of operands that an operation takes
 - Multiplication & addition operators have two arguments (arity == 2)



Overview of Expression Tree Processing Domain

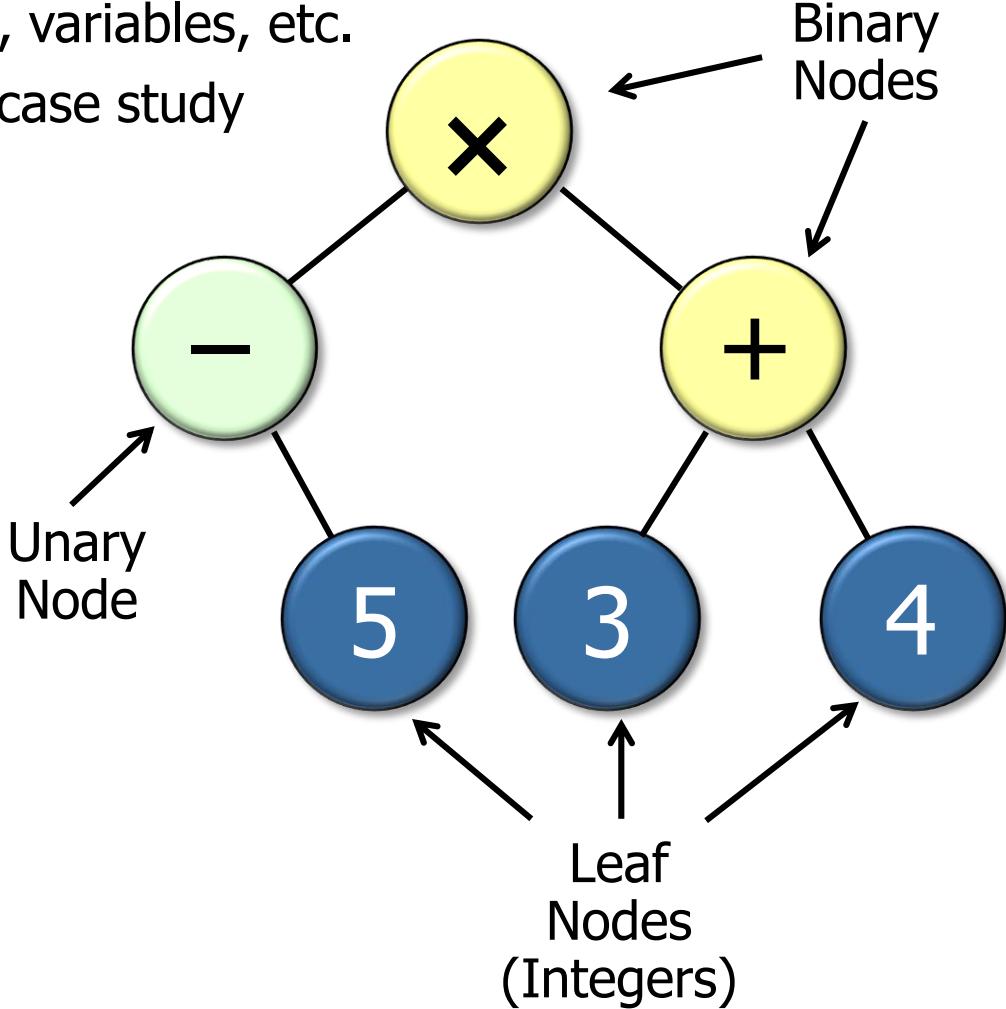
- Operators have different precedence levels, different associativities, & different arities, e.g.

- Precedence defines which operator to perform first to evaluate a mathematical expression
- Associativity determines how operators of the same level of precedence are grouped in the absence of parentheses
- Arity defines the # of operands that an operation takes
 - Multiplication & addition operators have two arguments (arity == 2)
 - The unary minus operator has one argument (arity == 1)



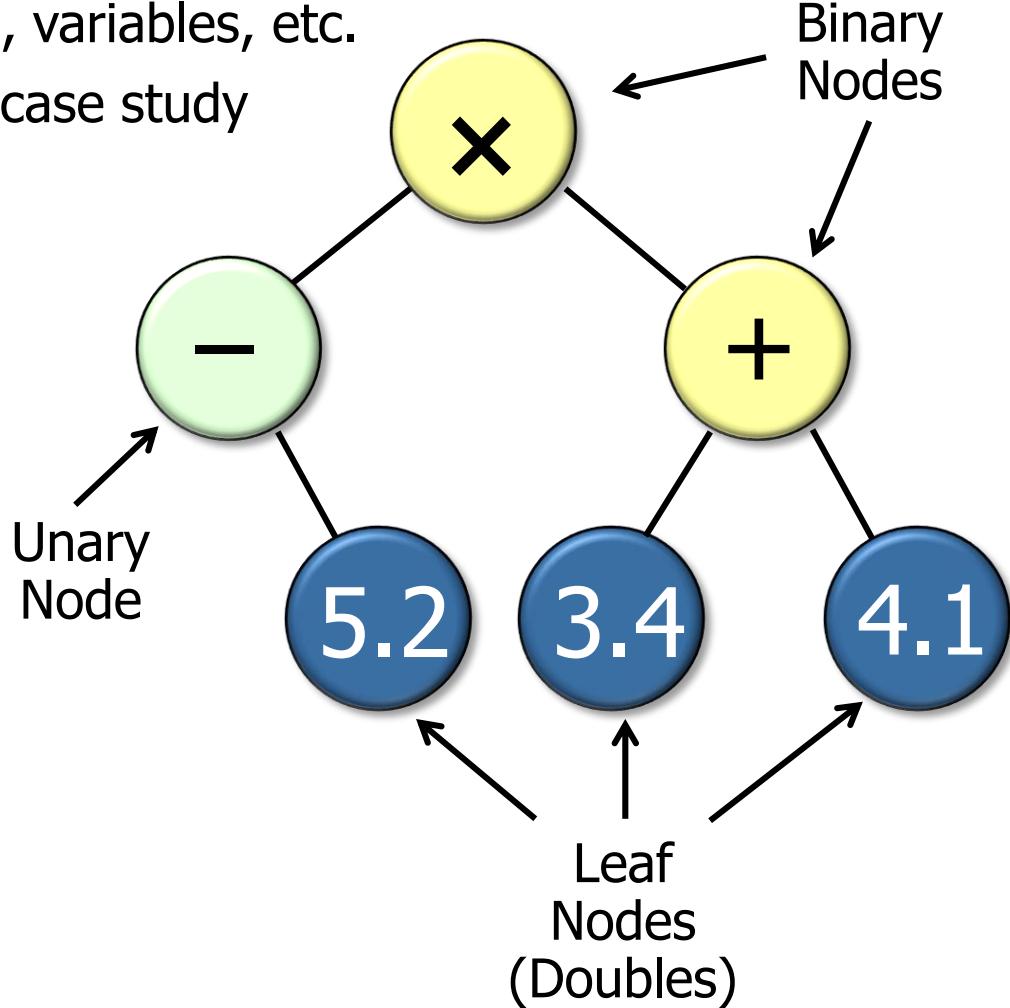
Overview of Expression Tree Processing Domain

- Operands can be integers, doubles, variables, etc.
 - We'll just handle integers in this case study



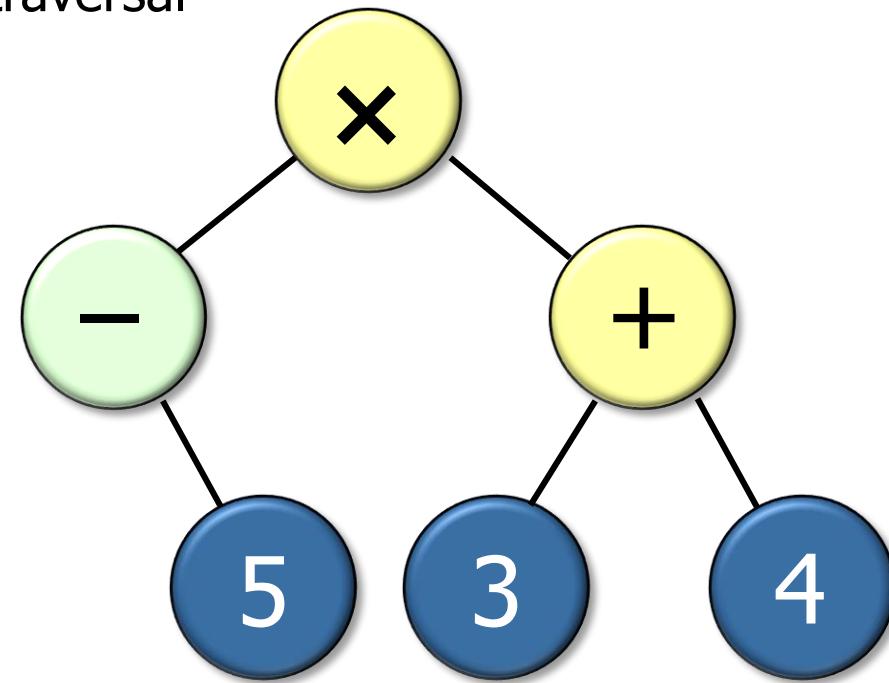
Overview of Expression Tree Processing Domain

- Operands can be integers, doubles, variables, etc.
 - We'll just handle integers in this case study
 - It's easy to extend the app to handle other types



Overview of Expression Tree Processing Domain

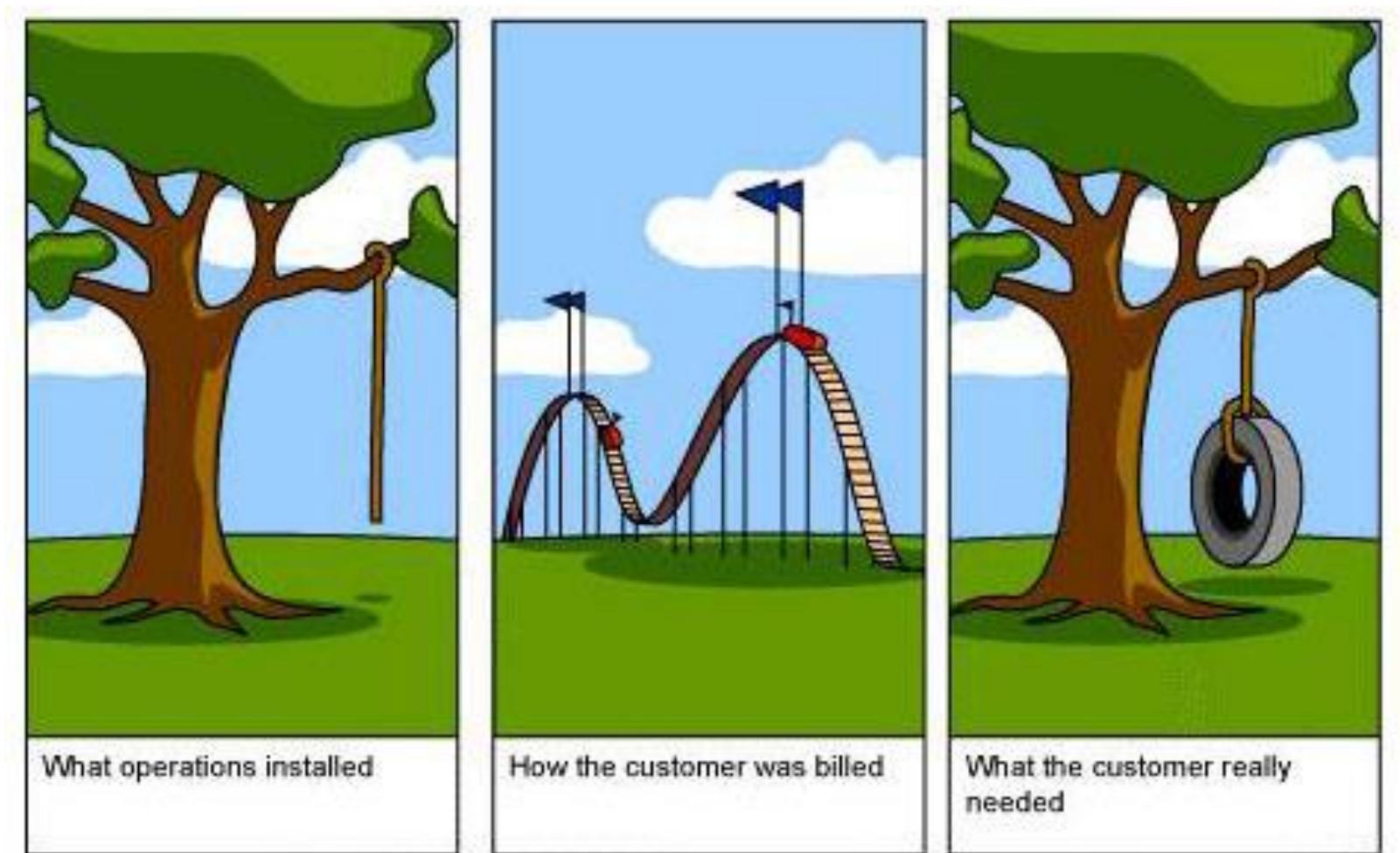
- Trees may be “evaluated” via different traversal orders, e.g.
 - “in-order traversal” = $-5 \times (3+4)$
 - “pre-order traversal” = $\times - 5 + 3 4$
 - “post-order traversal” = $5 - 3 4 + \times$
 - “level-order traversal” = $\times - + 5 3 4$



Functional & Non- Functional Requirements of the Case Study

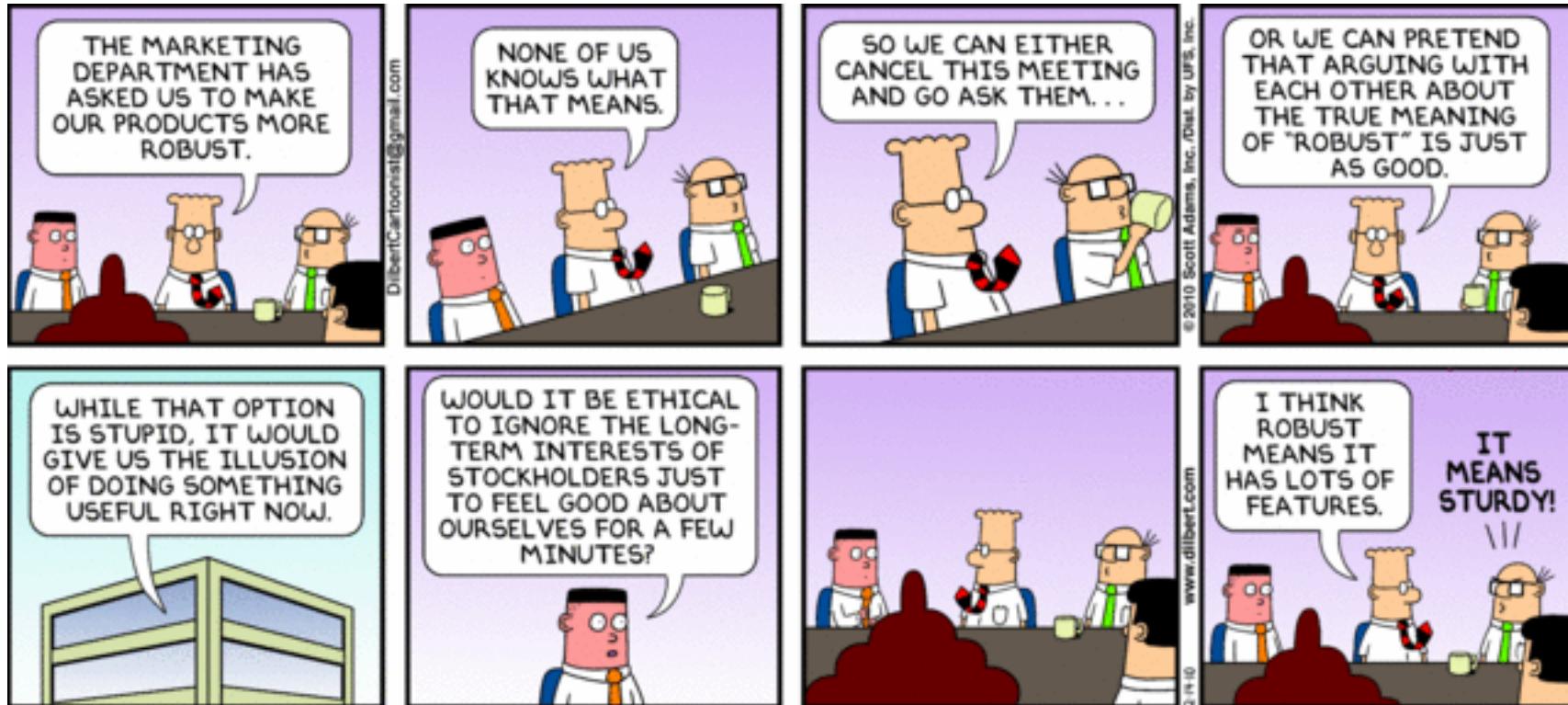
Functional & Non-Functional Requirements

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



Functional & Non-Functional Requirements

- A **Non-functional requirement** defines specifies 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

Case Study Functional Requirements

- “Succinct mode” – 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 ::= 'x' | '/'

add_sub ::= '+' | '-'

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

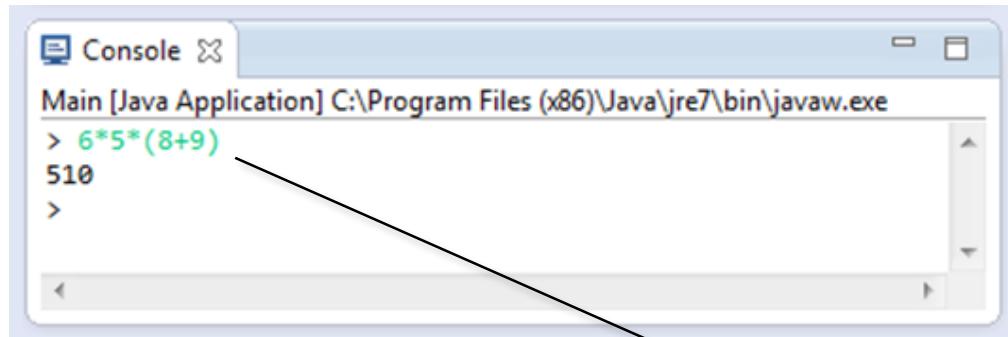
Case Study Functional Requirements

- Succinct mode can be command-line or GUI interface
 - In GUI version user presses Android buttons to enter expressions

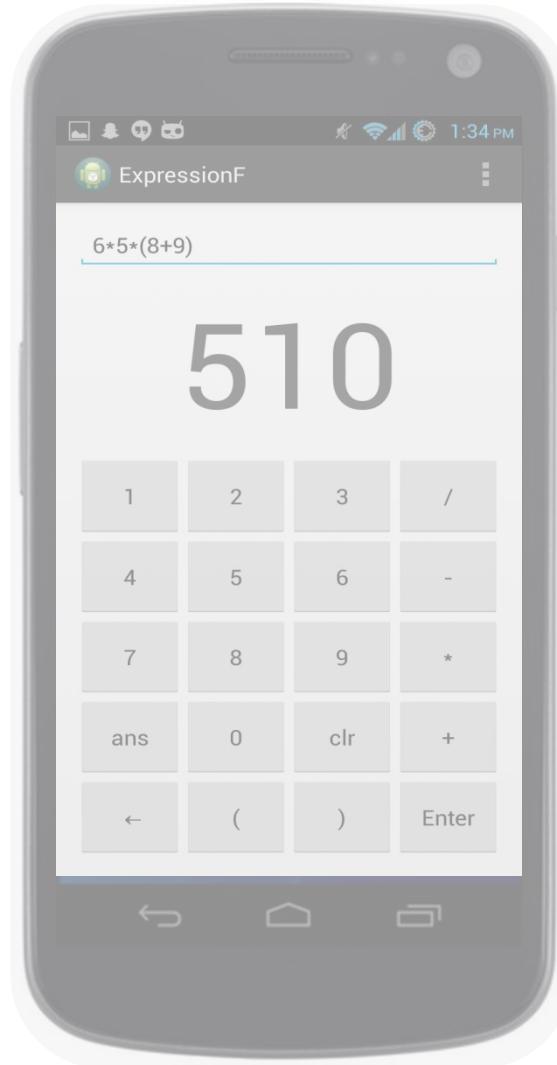


Case Study Functional Requirements

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



```
Console
Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe
> 6*5*(8+9)
510
>
```

A screenshot of a Java console window titled "Console". The window shows the command "6*5*(8+9)" being entered and its result "510" being displayed. A black arrow points from the text "In-fix expressions can contain parenthesized subexpressions" at the bottom of the slide towards the console window.

In-fix expressions can contain parenthesized subexpressions

Case Study Functional Requirements

- “Verbose mode” – Prompt user to enter command requests that control app behavior
 - The order of these command requests must follow a specific protocol

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

Case Study Functional Requirements

- “Verbose mode” – Prompt user to enter command requests that control app behavior
 - The order of these command requests must follow a specific protocol

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

Case Study Functional Requirements

- “Verbose mode” – Prompt user to enter command requests that control app behavior
 - The order of these command requests must follow a specific protocol

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

Case Study Functional Requirements

- “Verbose mode” – Prompt user to enter command requests that control app behavior
 - The order of these command requests must follow a specific protocol

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

Case Study Functional Requirements

- “Verbose mode” – Prompt user to enter command requests that control app behavior
 - The order of these command requests must follow a specific protocol

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

Case Study Functional Requirements

- “Verbose mode” – Prompt user to enter command requests that control app behavior
 - The order of these command requests must follow a specific protocol

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

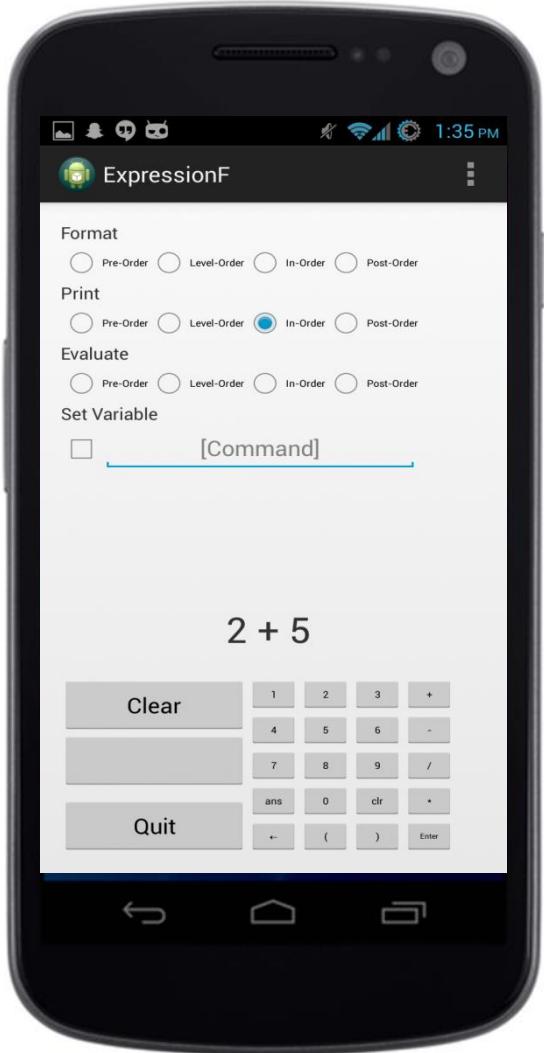
Case Study Functional Requirements

- “Verbose mode” – Prompt user to enter command requests that control app behavior
 - The order of these command requests must follow a specific protocol

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

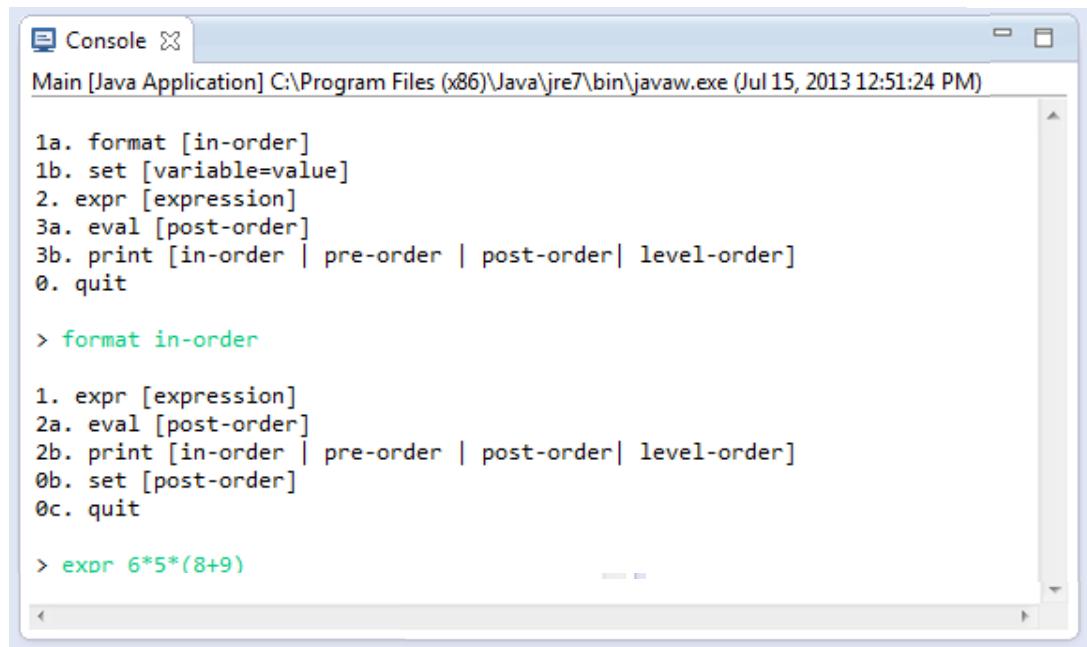
Case Study Functional Requirements

- Verbose mode can be accessed via
 - a GUI interface



Case Study Functional Requirements

- Verbose mode can be accessed via
 - a GUI interface
 - a command-line interface



Console

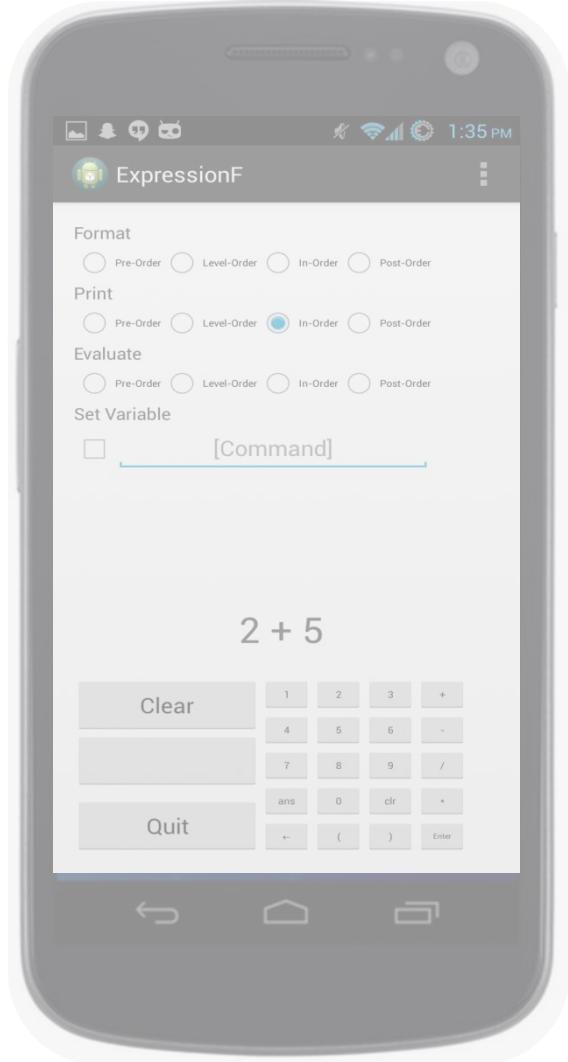
Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Jul 15, 2013 12:51:24 PM)

```
1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order| level-order]
0. quit

> format in-order

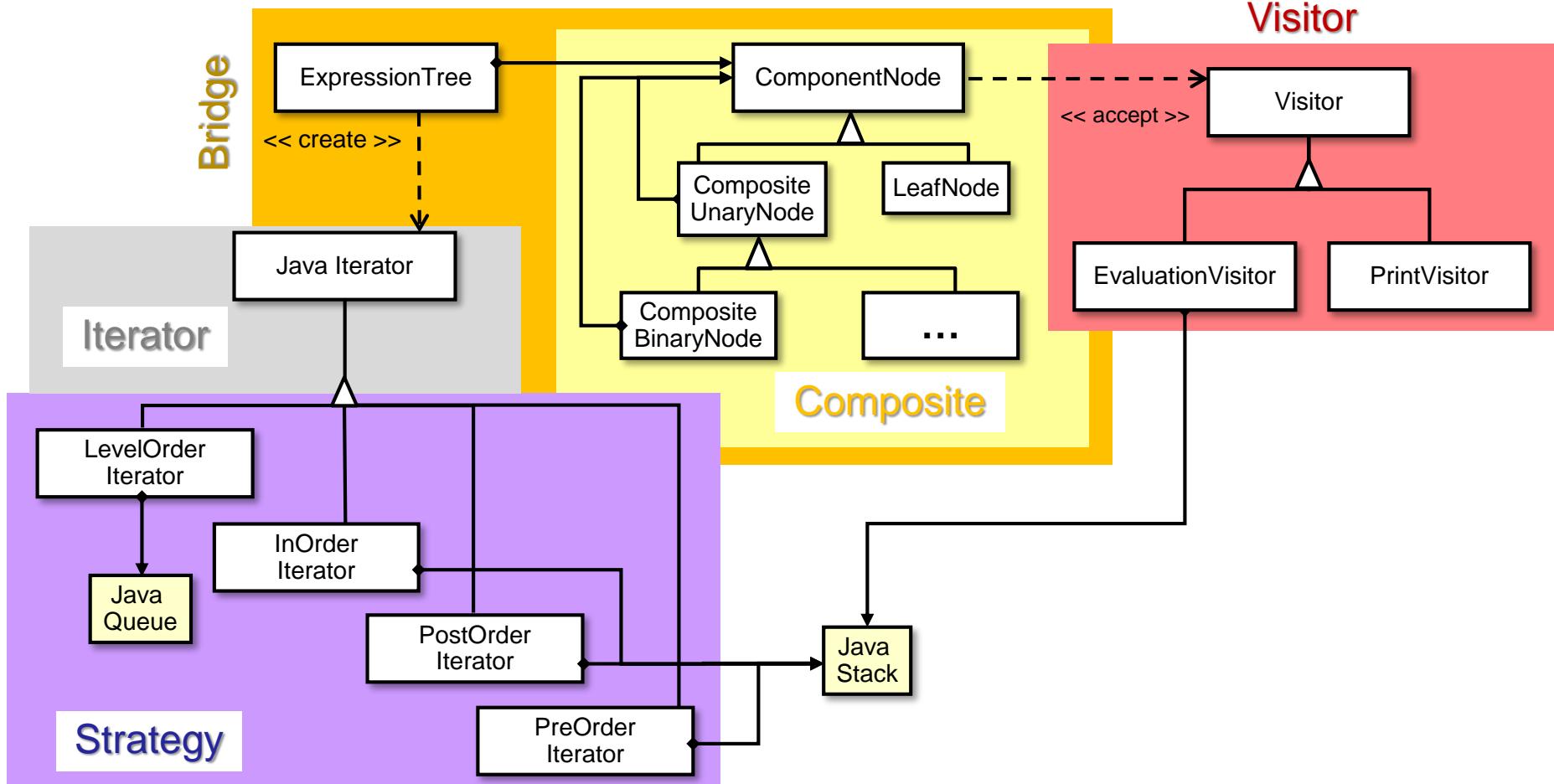
1. expr [expression]
2a. eval [post-order]
2b. print [in-order | pre-order | post-order| level-order]
0b. set [post-order]
0c. quit

> expr 6*5*(8+9)
```



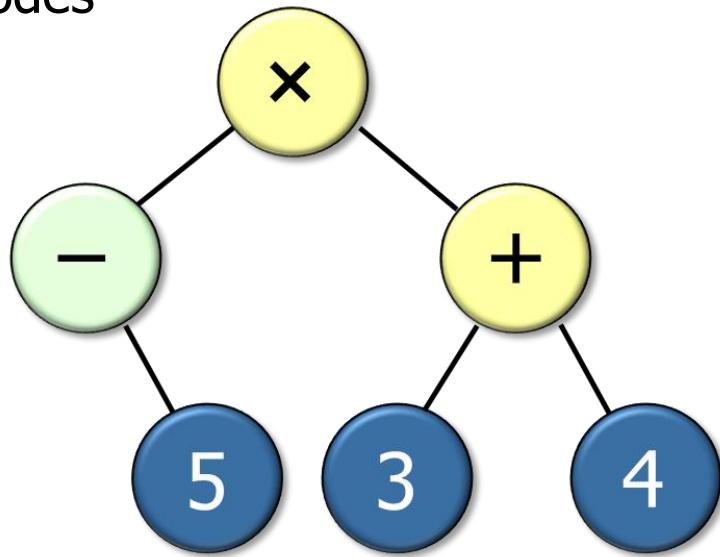
Case Study Non-Functional Requirements

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



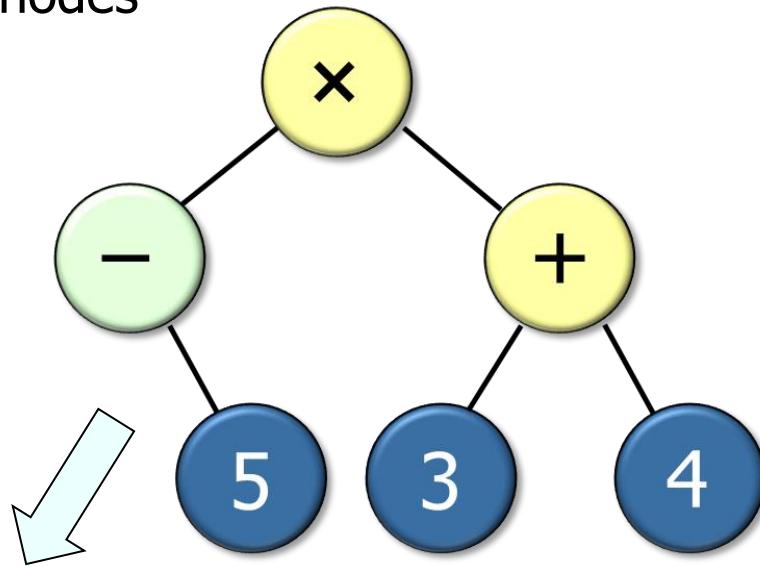
Case Study Non-Functional Requirements

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



Case Study Non-Functional Requirements

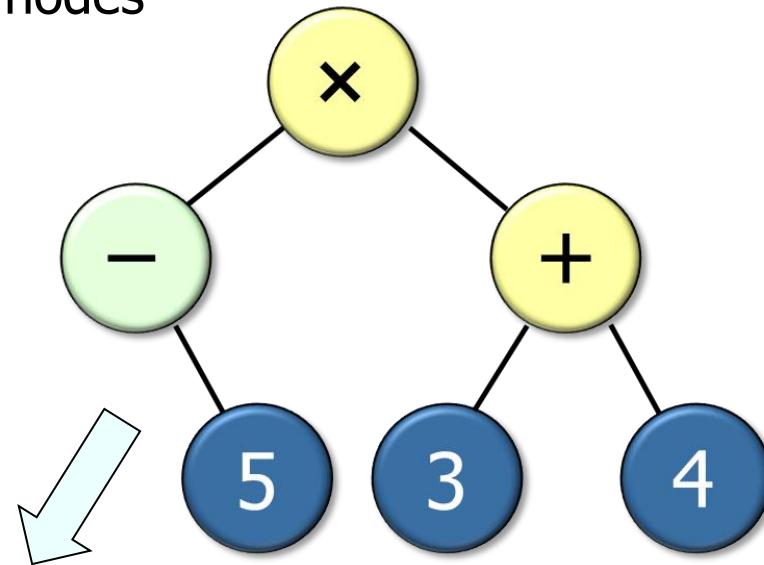
- Apply a pattern-oriented OO design to simplify extensibility & portability, e.g.
 - Add new operations on expression tree nodes without modifying the tree structure or implementation, e.g.
 - Print contents of expression tree in various traversal orders



- “In-order” traversal = $-5 \times (3+4)$
- “Pre-order” traversal = $\times - 5 + 3 4$
- “Post-order” traversal = $5 - 3 4 + \times$
- “Level-order” traversal = $\times - + 5 3 4$

Case Study Non-Functional Requirements

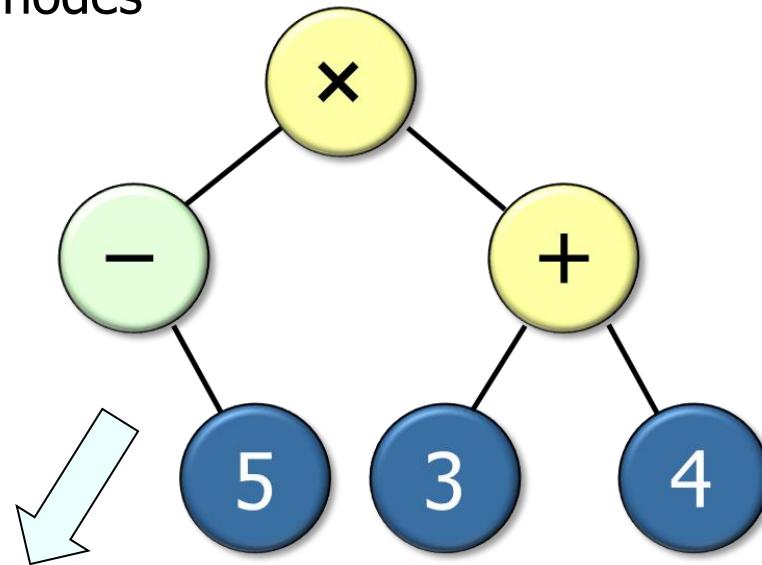
- Apply a pattern-oriented OO design to simplify extensibility & portability, e.g.
 - Add new operations on expression tree nodes without modifying the tree structure or implementation, e.g.
 - Print contents of expression tree in various traversal orders
 - Compute the "value" of the expression tree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



1. S = [5]	<code>push (node.getItem())</code>
2. S = [-5]	<code>push (-pop())</code>
3. S = [-5, 3]	<code>push (node.getItem())</code>
4. S = [-5, 3, 4]	<code>push (node.getItem())</code>
5. S = [-5, 7]	<code>push (pop() +pop())</code>
6. S = [-35]	<code>push (pop() *pop())</code>

Case Study Non-Functional Requirements

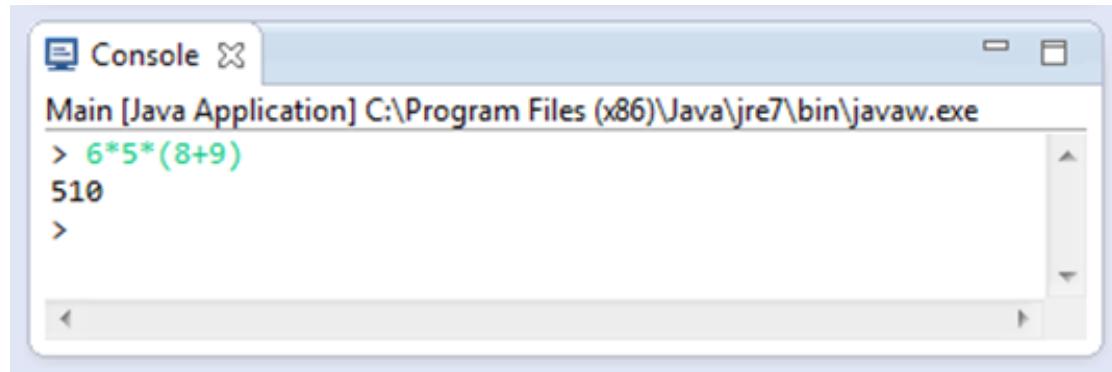
- Apply a pattern-oriented OO design to simplify extensibility & portability, e.g.
 - Add new operations on expression tree nodes without modifying the tree structure or implementation, e.g.
 - Print contents of expression tree in various traversal orders
 - Compute the “value” of the expression tree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



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

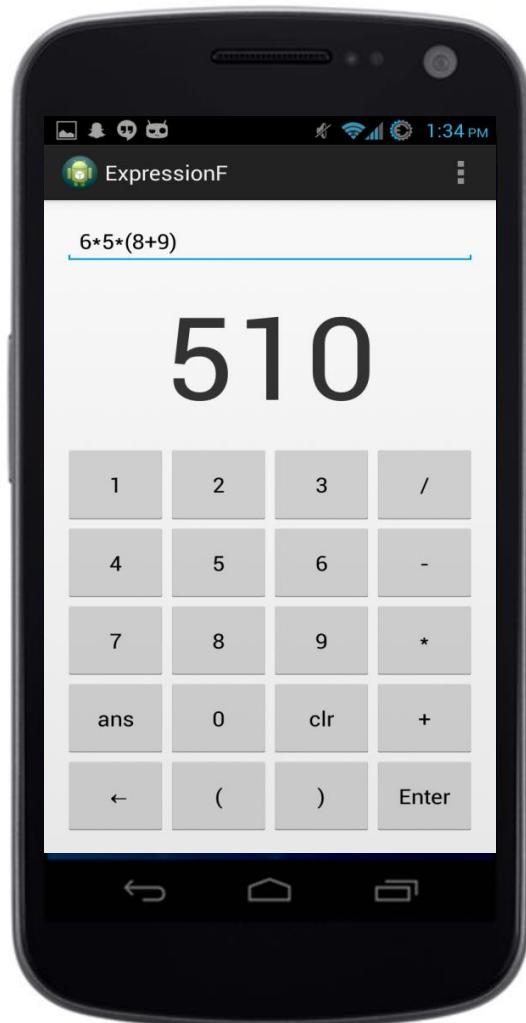
Case Study Non-Functional Requirements

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



A screenshot of a Java console window titled "Console". The title bar shows "Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe". The console output shows the expression $6*5*(8+9)$ being entered, followed by the result 510, and then a prompt greater than sign (>).

```
Console
Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe
> 6*5*(8+9)
510
>
```



Putting All the Pieces Together

Putting All the Pieces Together

- The expression tree processing app provides 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 operation implementations	Command
Inflexible creation of variabilities	Factory Method
Inflexible expression tree traversal	Iterator
Obtrusive behavior changes	Strategy
Non-extensible tree operations	Visitor
Incorrect operation ordering	State
Non-extensible operating modes	Template Method
Minimizing global variable liabilities	Singleton

Putting All the Pieces Together

- The expression tree processing app provides a realistic case study of how to apply GoF patterns
 - All the case study code is written in Java (~4K LOC & 57 classes)

douglasraigschmidt updates Latest commit c99f134 on Nov 23, 2017

..

AndroidGUI	updates	4 months ago
CommandLine	updates	4 months ago
original	updates	4 months ago
README.md	updates	4 months ago

README.md

This folder contains source code for the following apps that implement pattern-oriented variants of ExpressionTree

original -- This folder provides the original implementation of this app from my [Design Patterns in Java](#) LiveLesson tutorial. This version compiles/runs in Eclipse configured with Android.

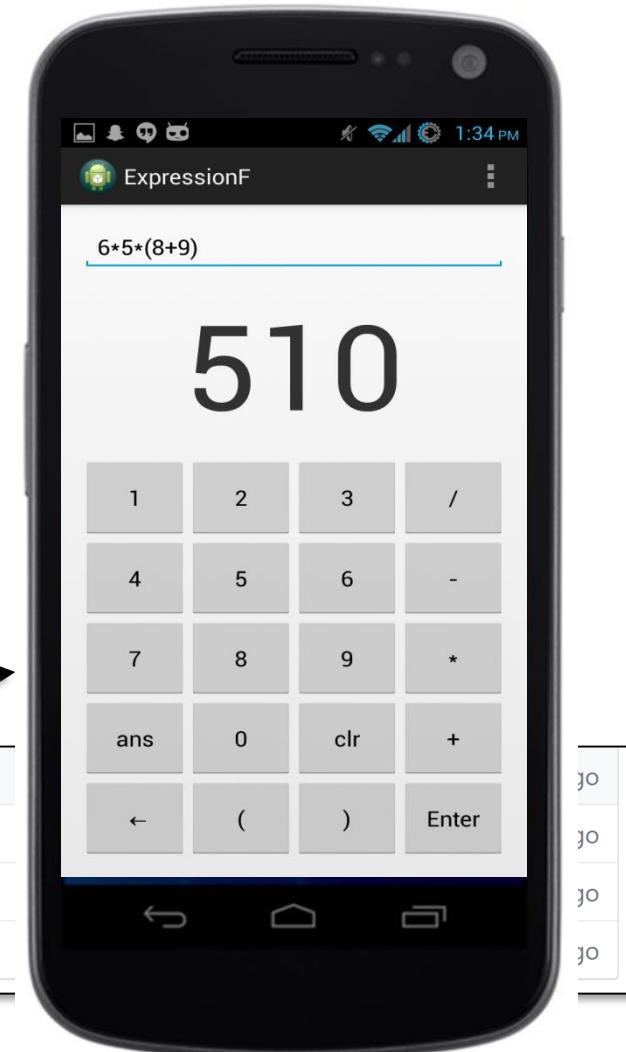
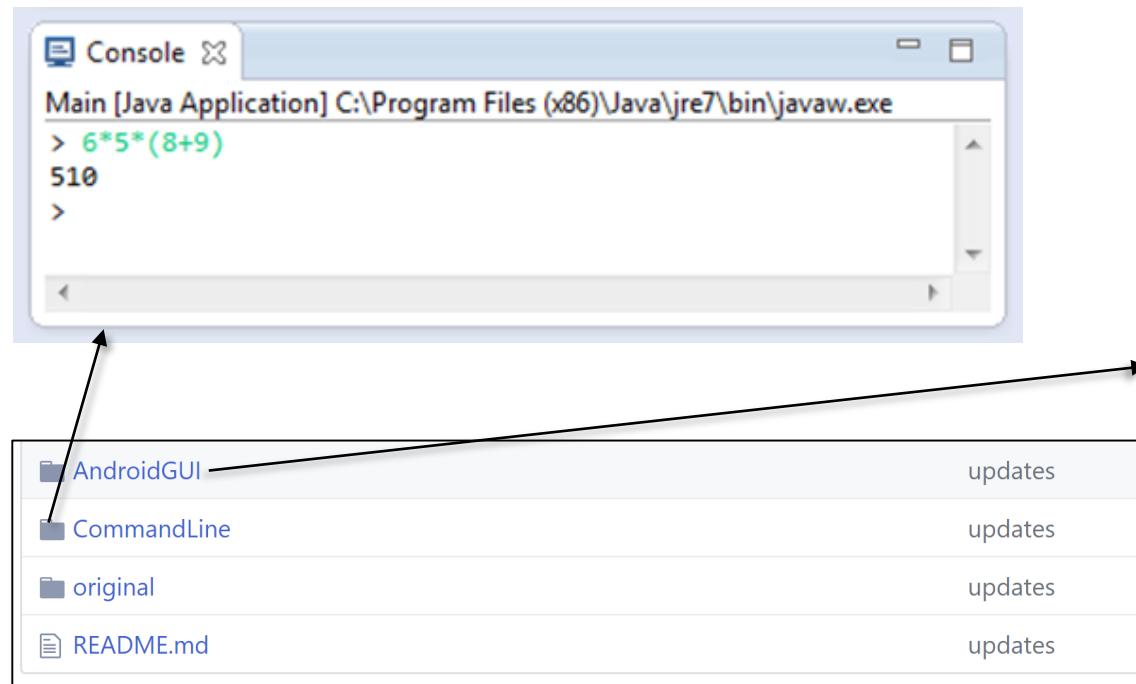
CommandLine -- This folder provides an updated command-line-only version of my LiveLessons tutorial that uses Java 8 features, such as lambdas and method references. This version compiles/runs in IntelliJ.

AndroidGUI -- This folder provides an updated Android GUI-only version of my LiveLessons tutorial that uses Java 8 features, such as lambdas and method references. This version compiles/runs in Android Studio.

See github.com/douglasraigschmidt/LiveLessons/tree/master/ExpressionTree

Putting All the Pieces Together

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



See github.com/douglascraigschmidt/LiveLessons/tree/master/ExpressionTree

**End of Overview of the
Expression Tree Processing
App Case Study**

Evaluating the Object-Oriented Design of the Expression Tree Processing App

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

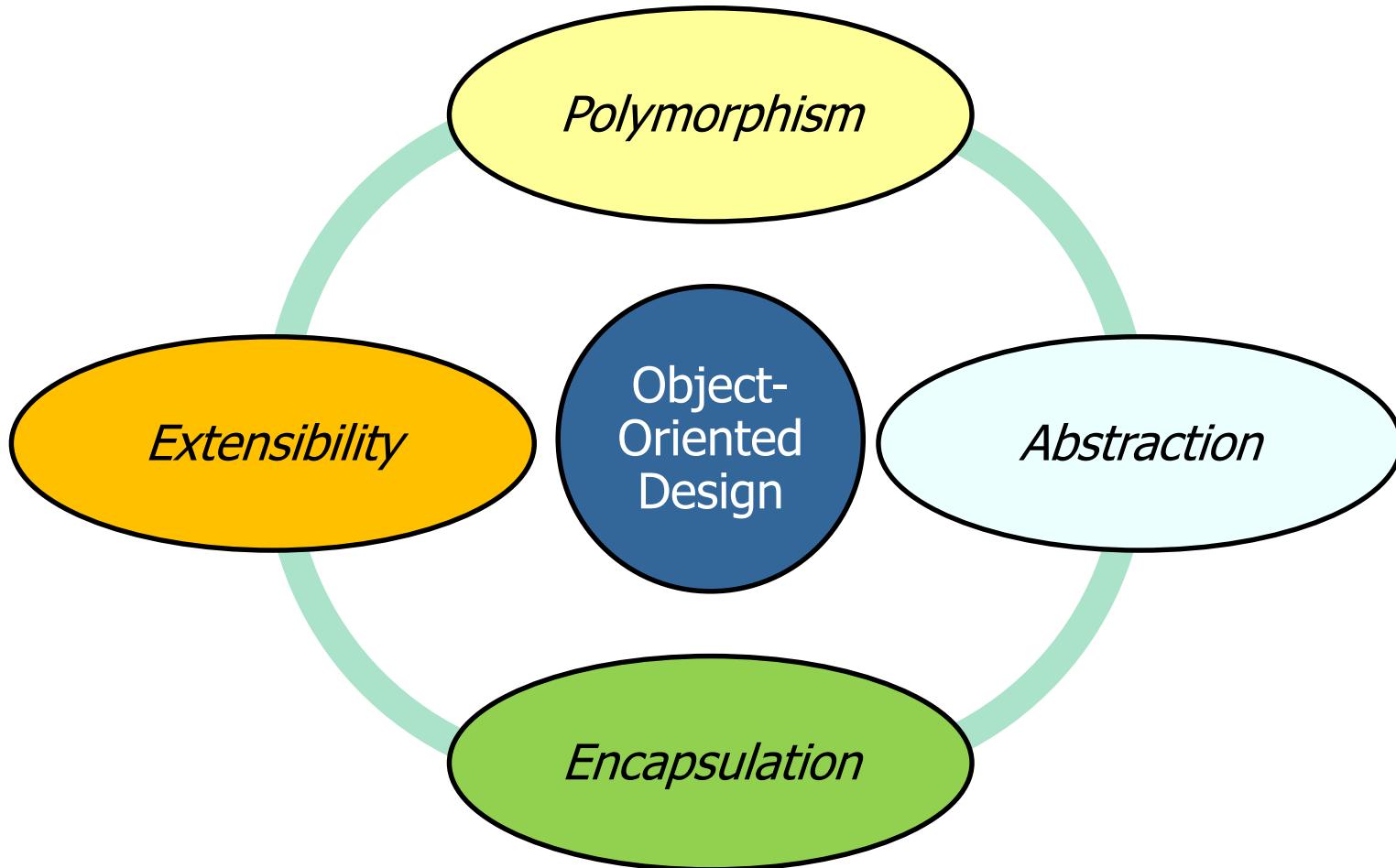
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



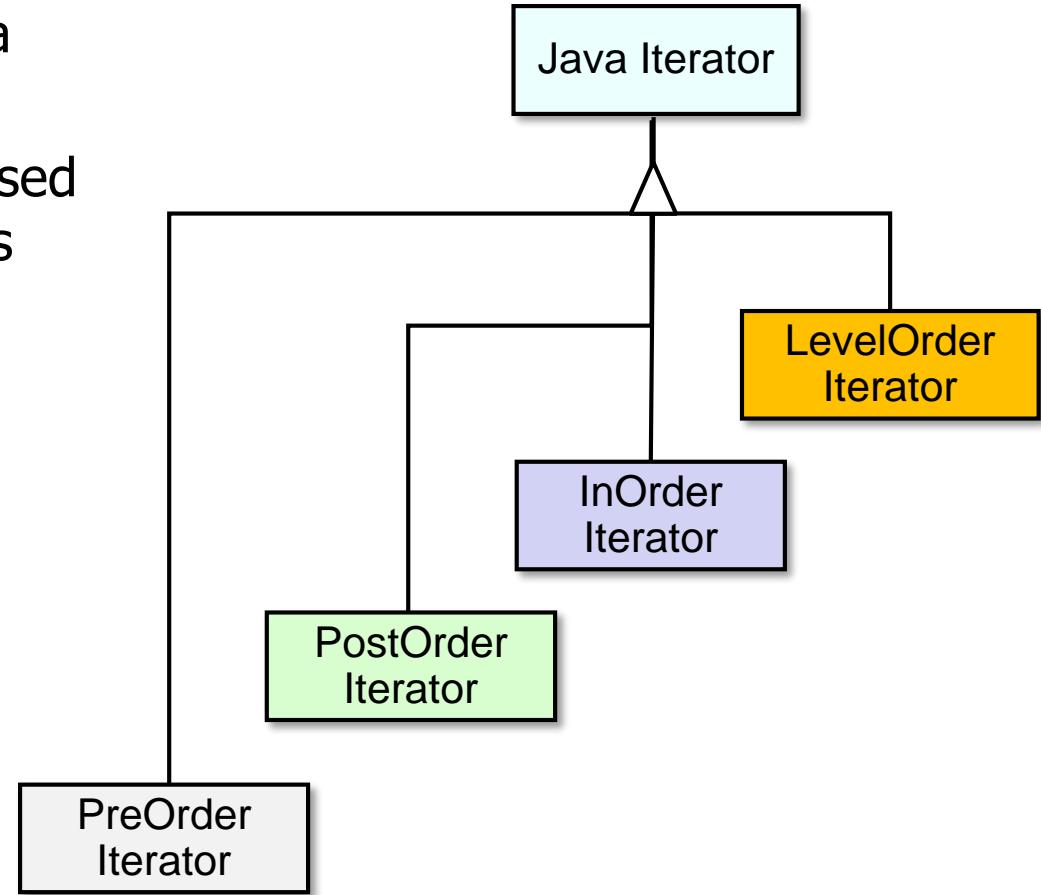
Lesson Intro

- Object-oriented design (OOD) is a method of planning a system of interacting objects to solve software problem(s)



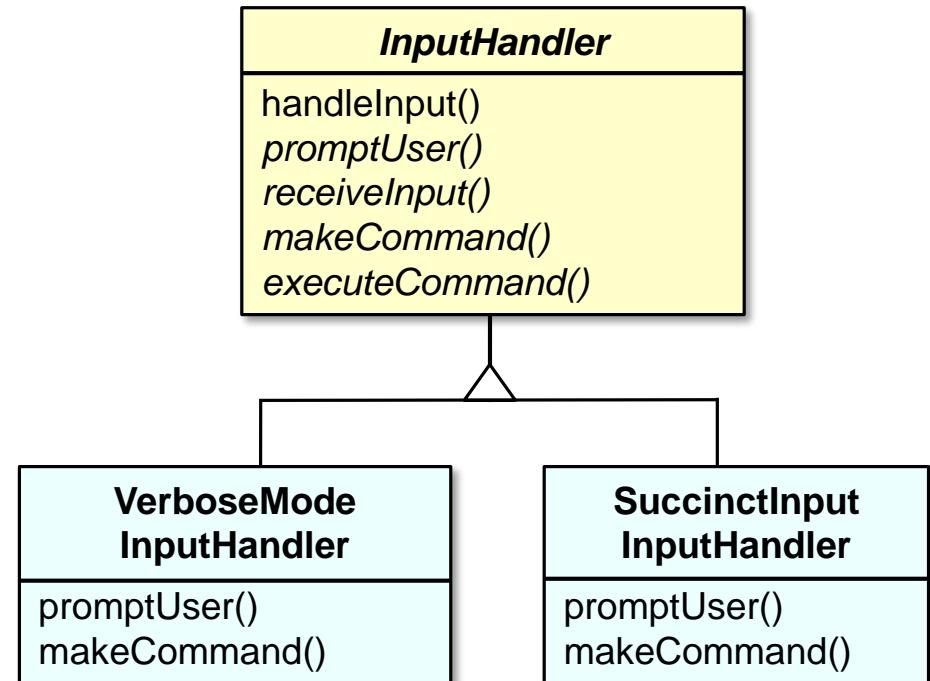
Lesson Intro

- Object-oriented design (OOD) is a method of planning a system of interacting objects to solve software problem(s)
- OOD employs “hierarchical data abstraction”
 - Components are designed based on stable *class* & *object* roles & relationships
 - Rather than functions corresponding to actions



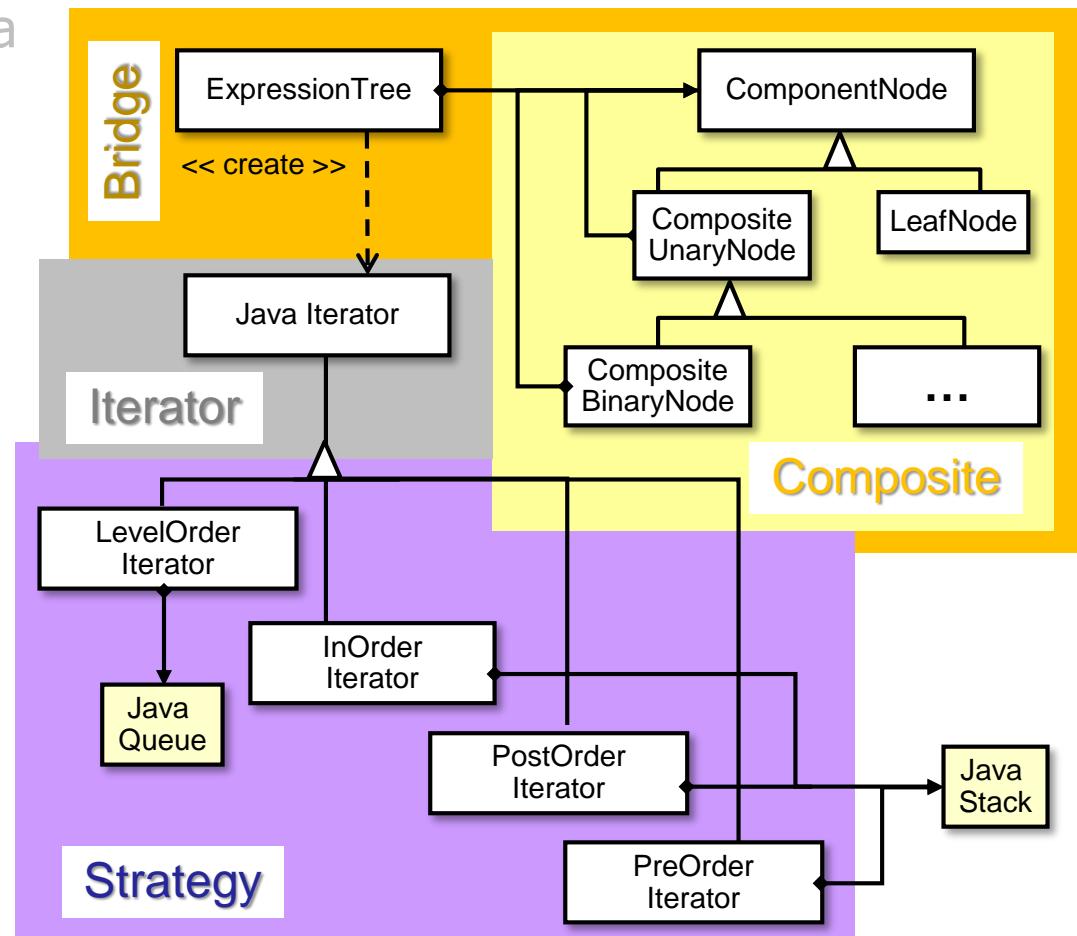
Lesson Intro

- Object-oriented design (OOD) is a method of planning a system of interacting objects to solve software problem(s)
- OOD employs “hierarchical data abstraction”
- It also associates actions with specific objects and/or classes of objects
 - Emphasize *high cohesion* & *low coupling*



Lesson Intro

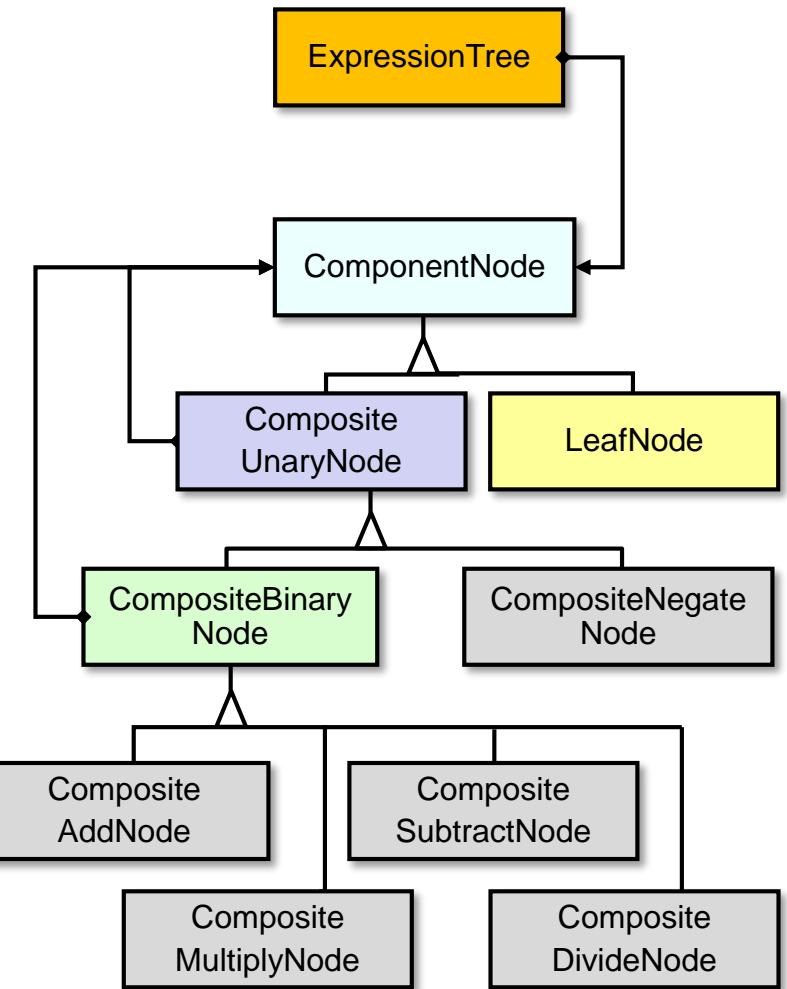
- Object-oriented design (OOD) is a method of planning a system of interacting objects to solve software problem(s)
- OOD employs “hierarchical data abstraction”
- It also associates actions with specific objects and/or classes of objects
- Well-designed OO programs group classes & objects via *patterns* & combine them to form *frameworks*



Learning Objectives

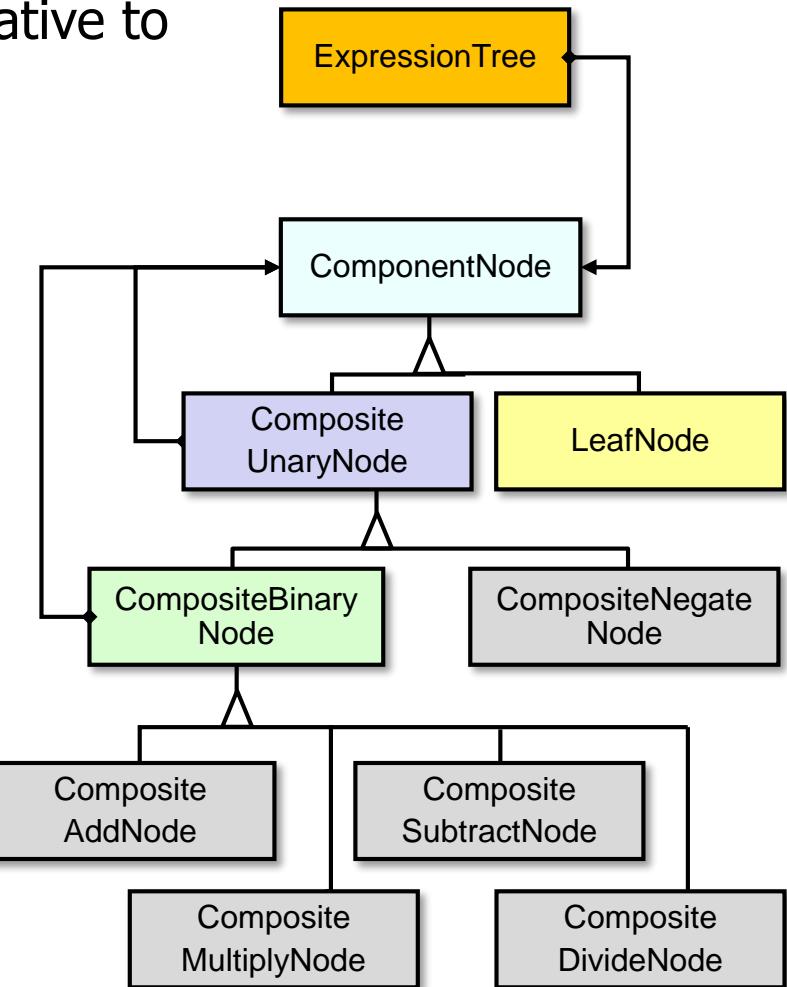
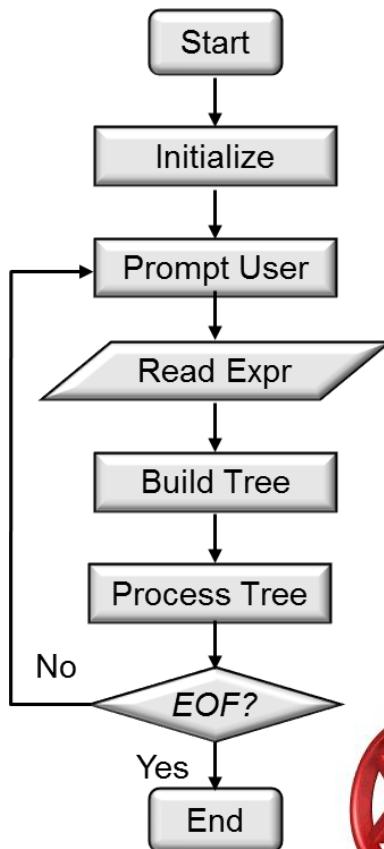
Learning Objectives

- Understand the OO design of the expression tree processing app



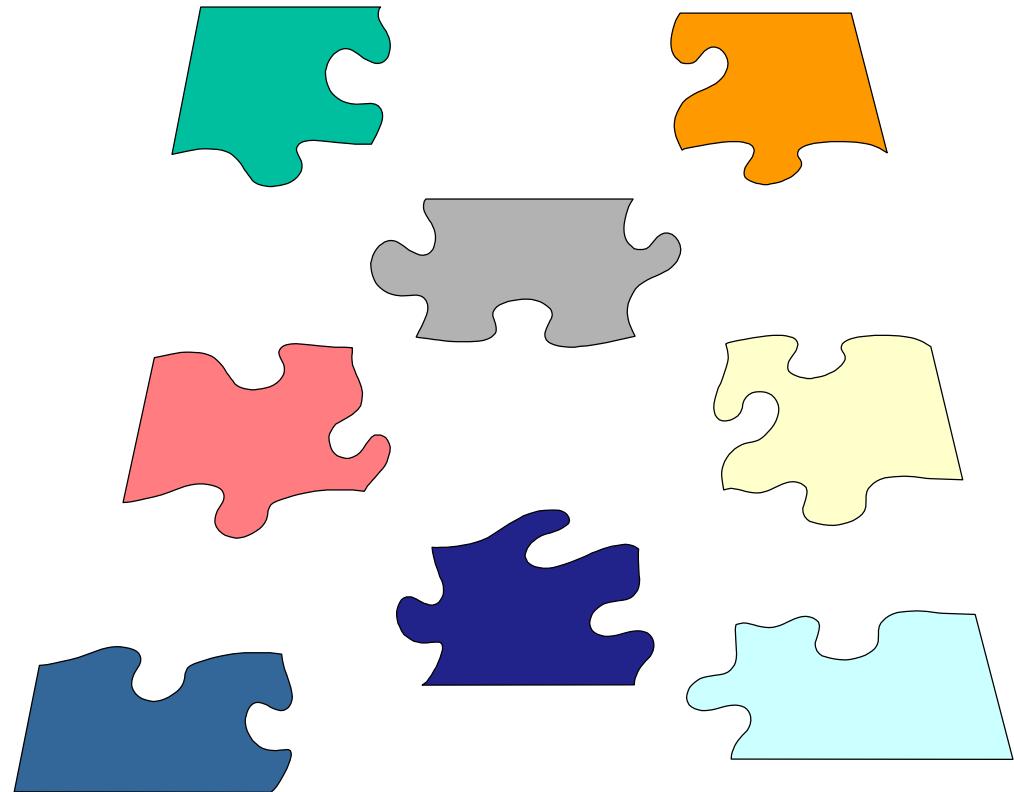
Learning Objectives

- Understand the OO design of the expression tree processing app
- Evaluate the pros & cons of OO design relative to algorithmic decomposition



Learning Objectives

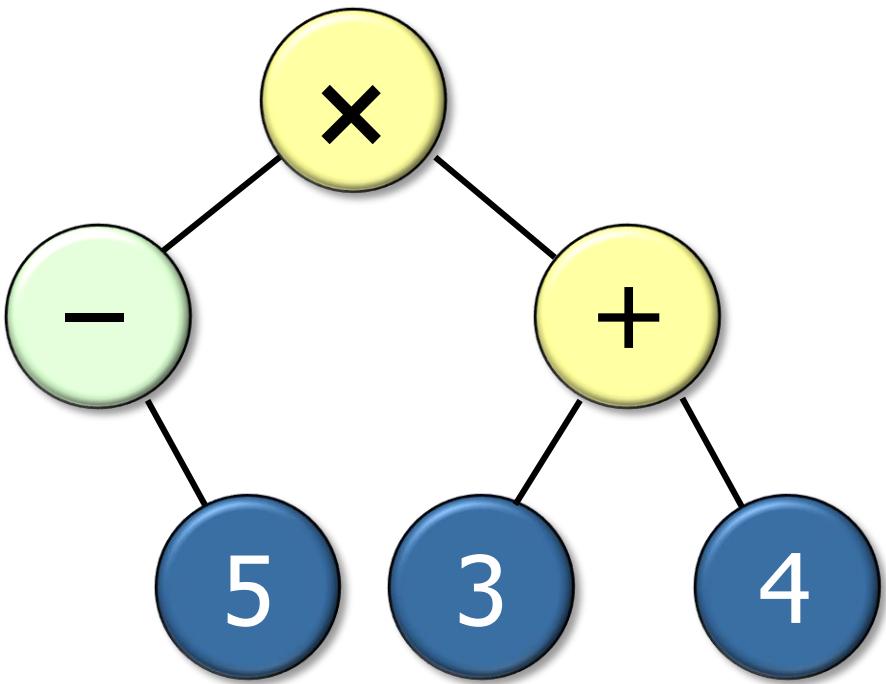
- Understand the OO design of the expression tree processing app
- Evaluate the pros & cons of OO design relative to algorithmic decomposition
- Put all the pieces together



OO Design of Expression Tree Processing App

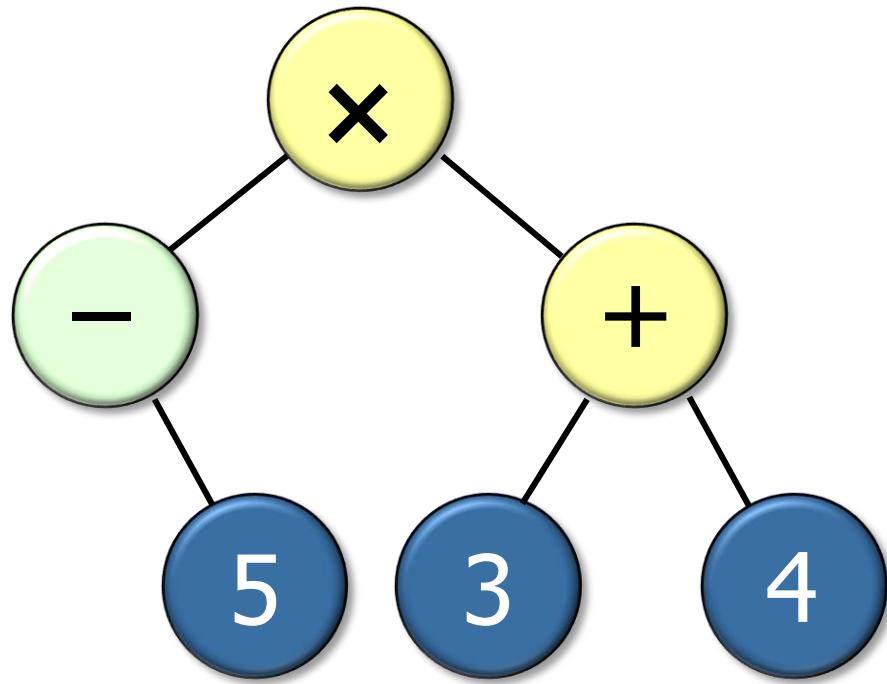
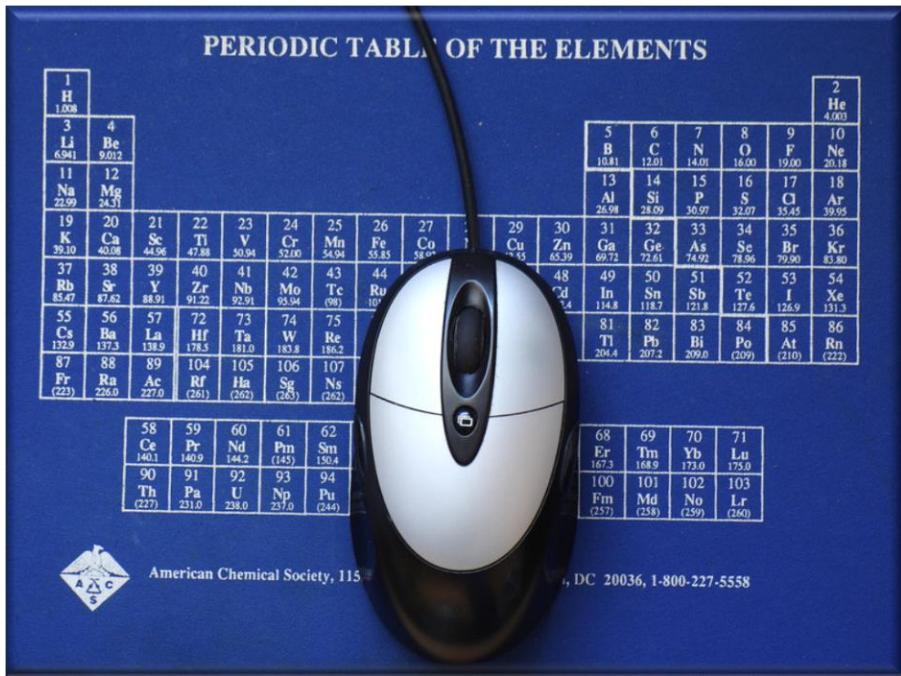
OO Design of Expression Tree Processing App

- Create an OO design based on modeling classes & objects in “expression tree” domain



OO Design of Expression Tree Processing App

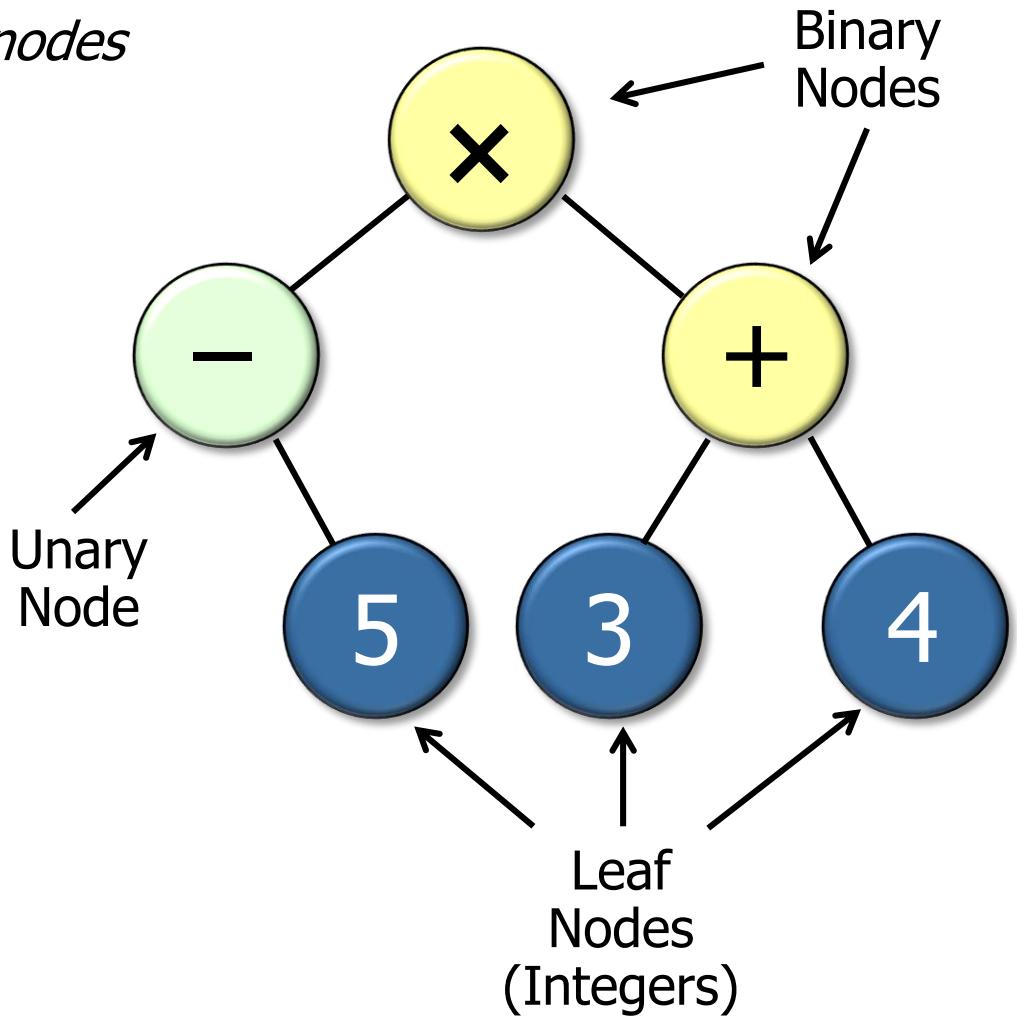
- Conduct *Scope, Commonality, & Variability* analysis to determine stable APIs & variable extension points



See www.dre.Vanderbilt.edu/~schmidt/PDF/Commonality_Variability.pdf

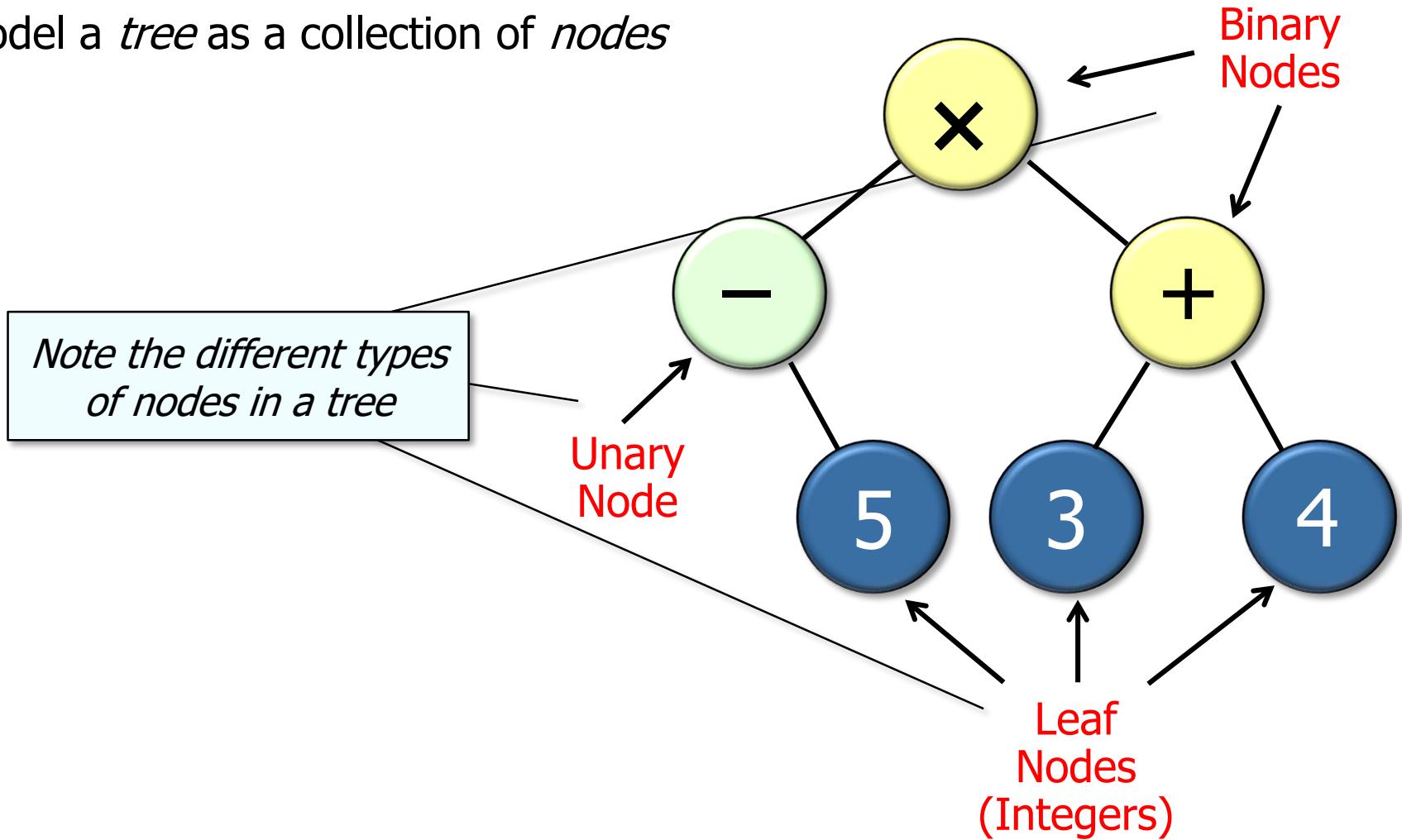
OO Design of Expression Tree Processing App

- Conduct *Scope, Commonality, & Variability* analysis to determine stable APIs & variable extension points
 - Model a *tree* as a collection of *nodes*



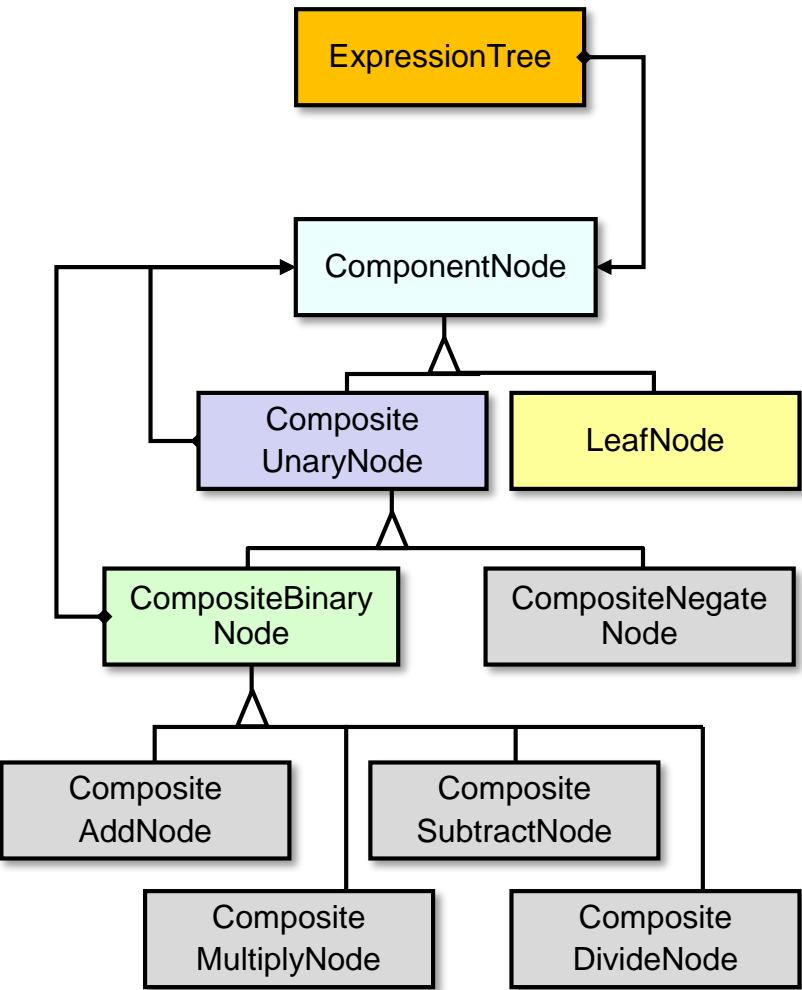
OO Design of Expression Tree Processing App

- Conduct *Scope, Commonality, & Variability* analysis to determine stable APIs & variable extension points
 - Model a *tree* as a collection of *nodes*



OO Design of Expression Tree Processing App

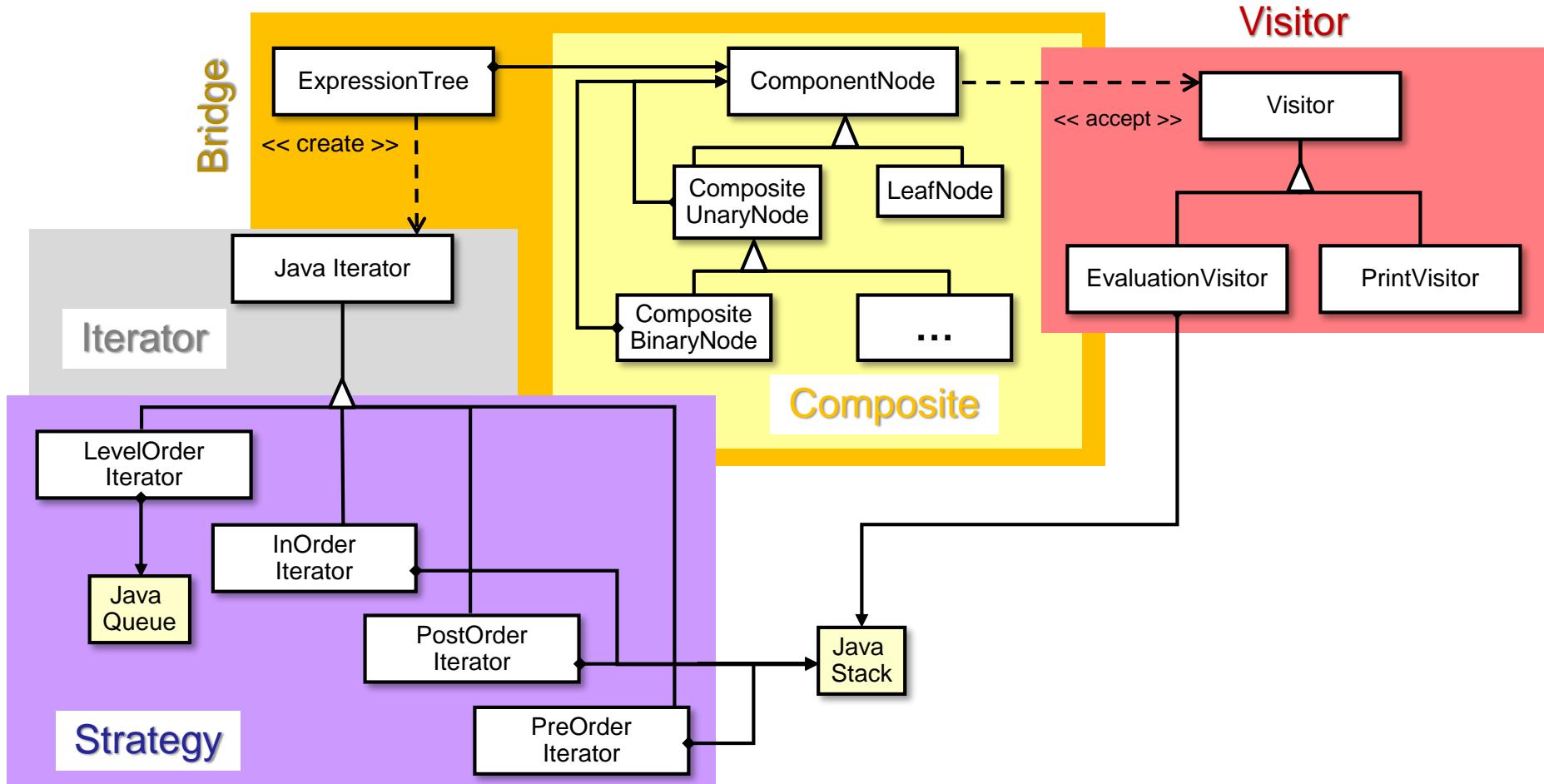
- Conduct *Scope, Commonality, & Variability* analysis to determine stable APIs & variable extension points
 - Model a *tree* as a collection of *nodes*
 - Represent *nodes* as class hierarchy, capturing properties of each node
 - e.g., the “arities” (binary & unary nodes)



See en.wikipedia.org/wiki/Arity

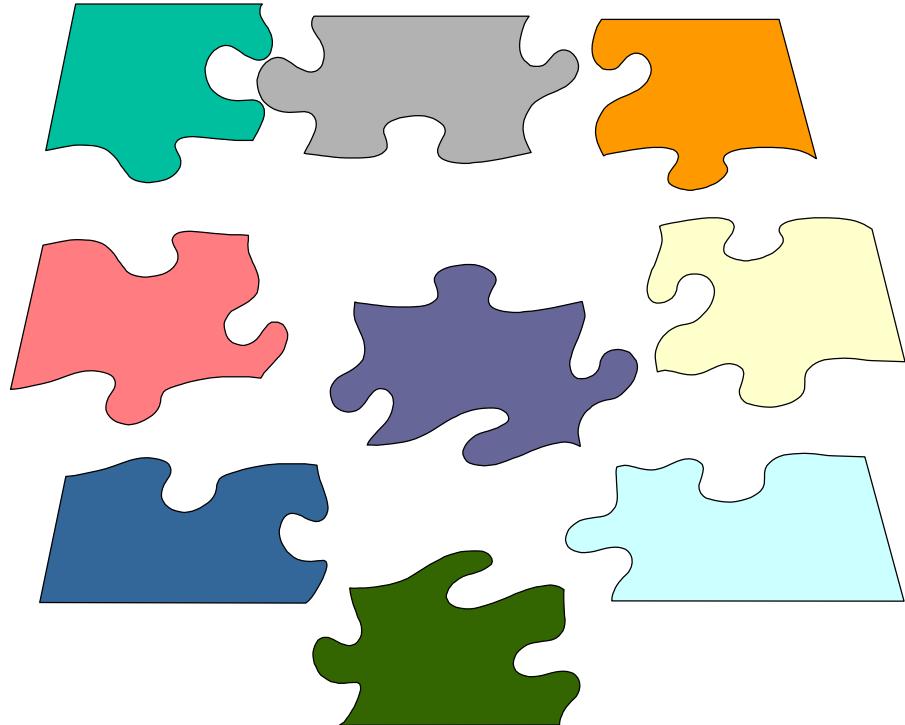
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes



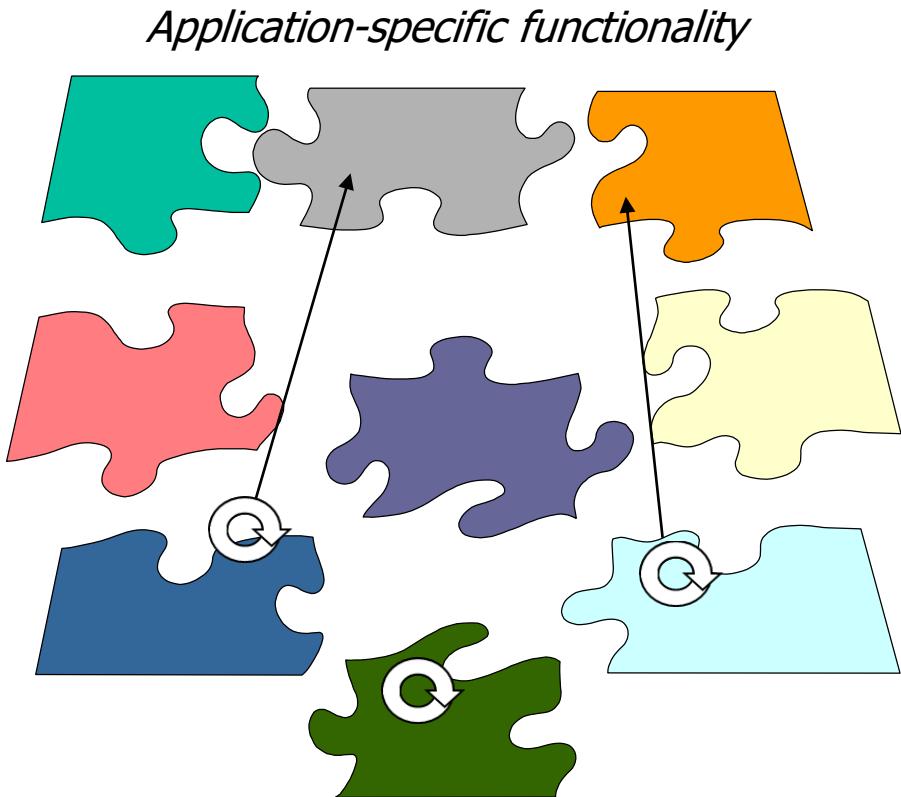
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications



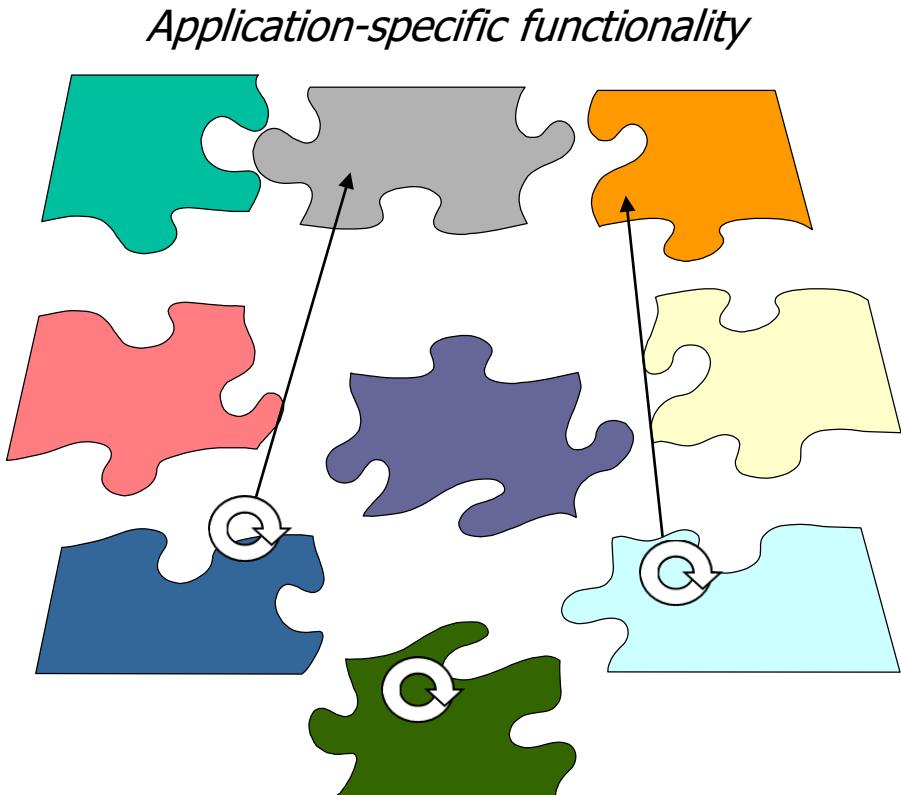
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse



OO Design of Expression Tree Processing App

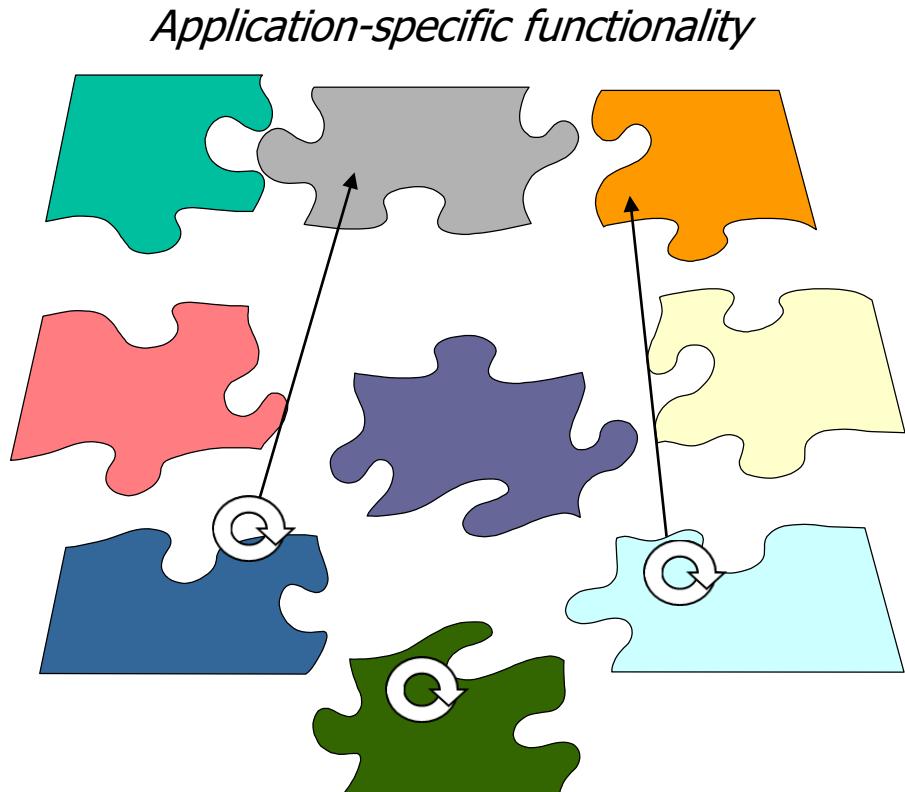
- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)



See en.wikipedia.org/wiki/Inversion_of_control

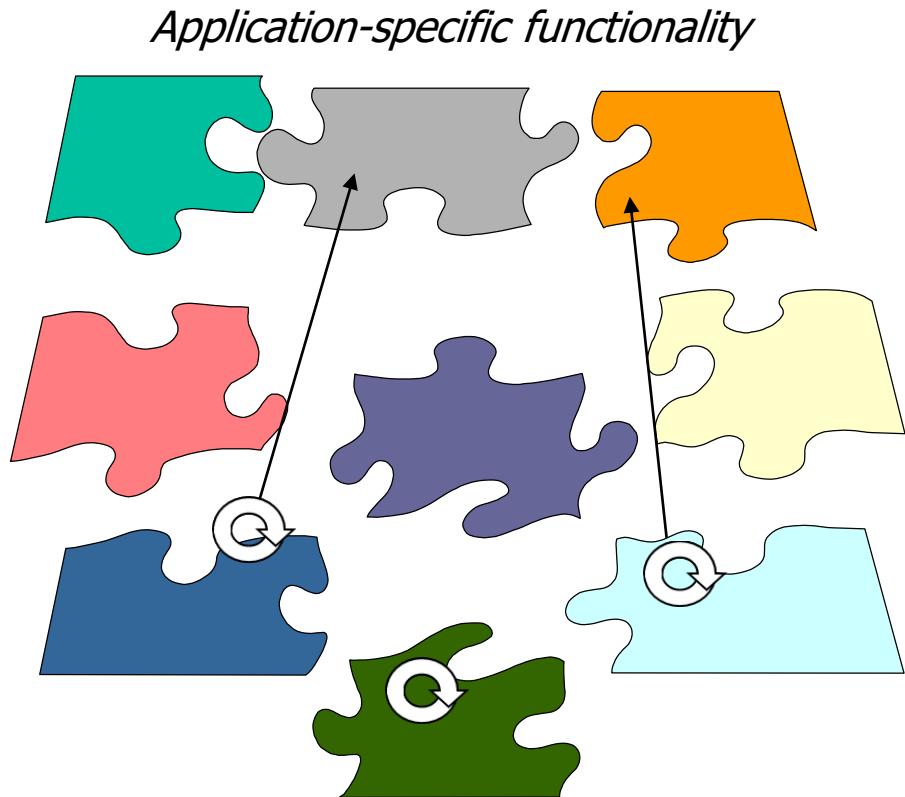
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Framework controls the main execution thread



OO Design of Expression Tree Processing App

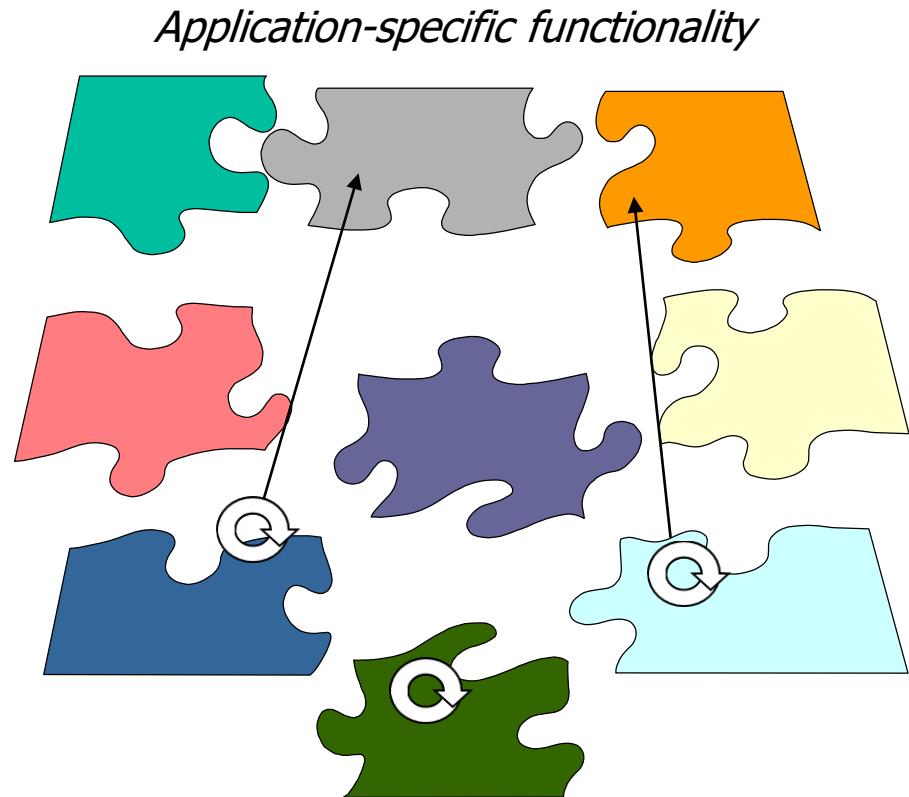
- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Framework controls the main execution thread
 - Decides how & when to run application code via callbacks



See [en.wikipedia.org/wiki/Callback_\(computer_programming\)](https://en.wikipedia.org/wiki/Callback_(computer_programming))

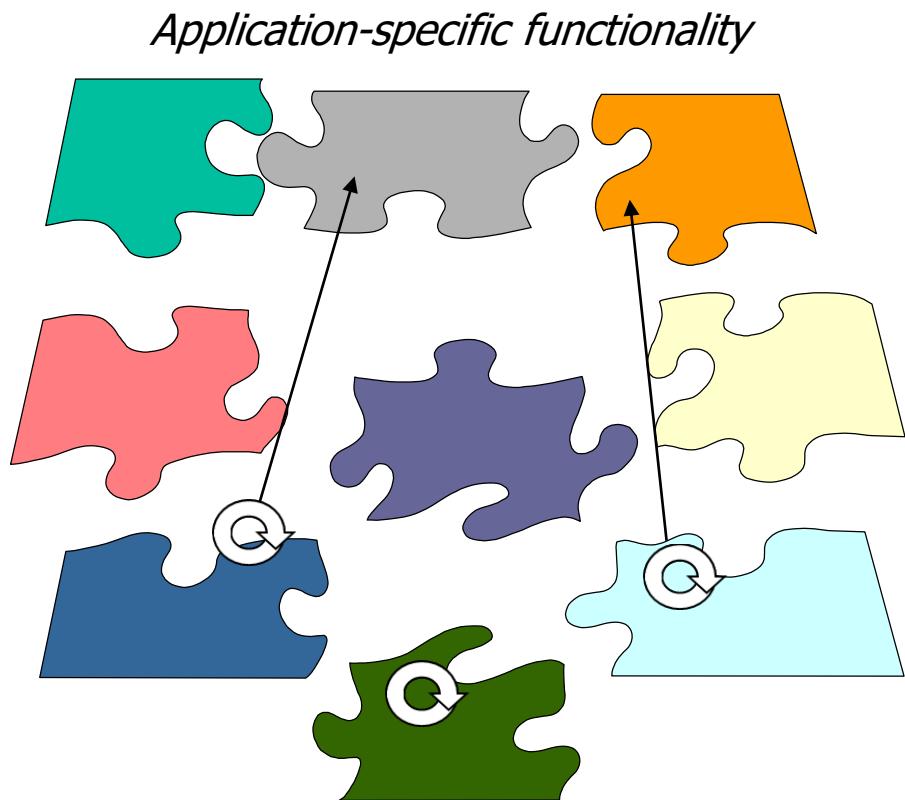
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Framework controls the main execution thread
 - Decides how & when to run application code via callbacks
 - e.g., an Android looper dispatches a handler, which then dispatches a runnable



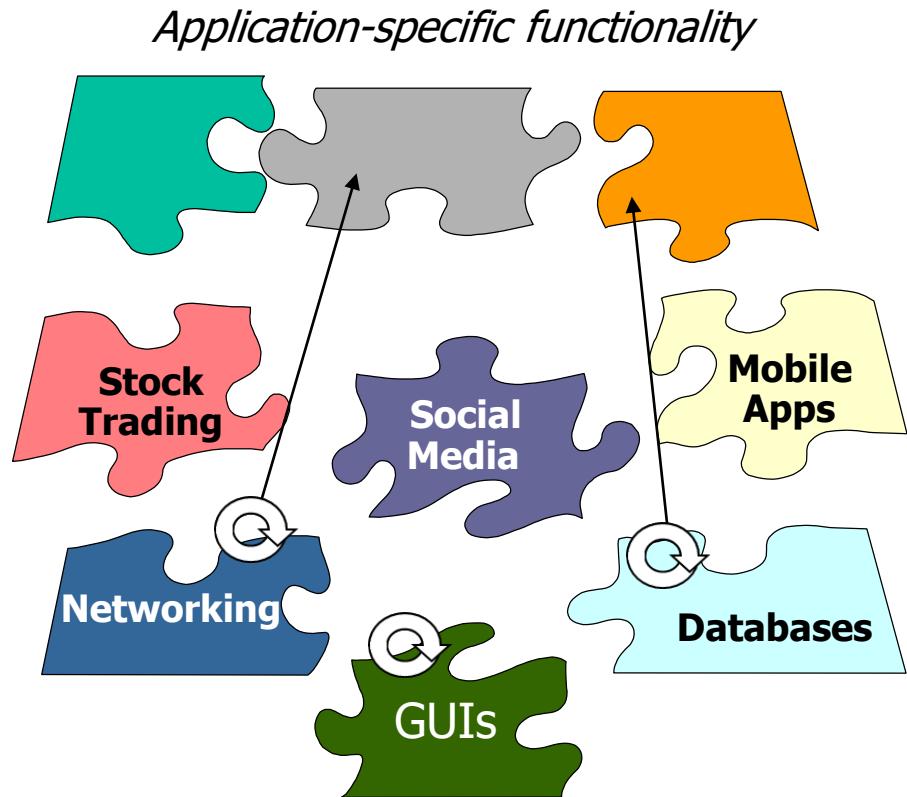
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Framework controls the main execution thread
 - Decides how & when to run application code via callbacks
 - IoC is often called “The Hollywood Principle”



OO Design of Expression Tree Processing App

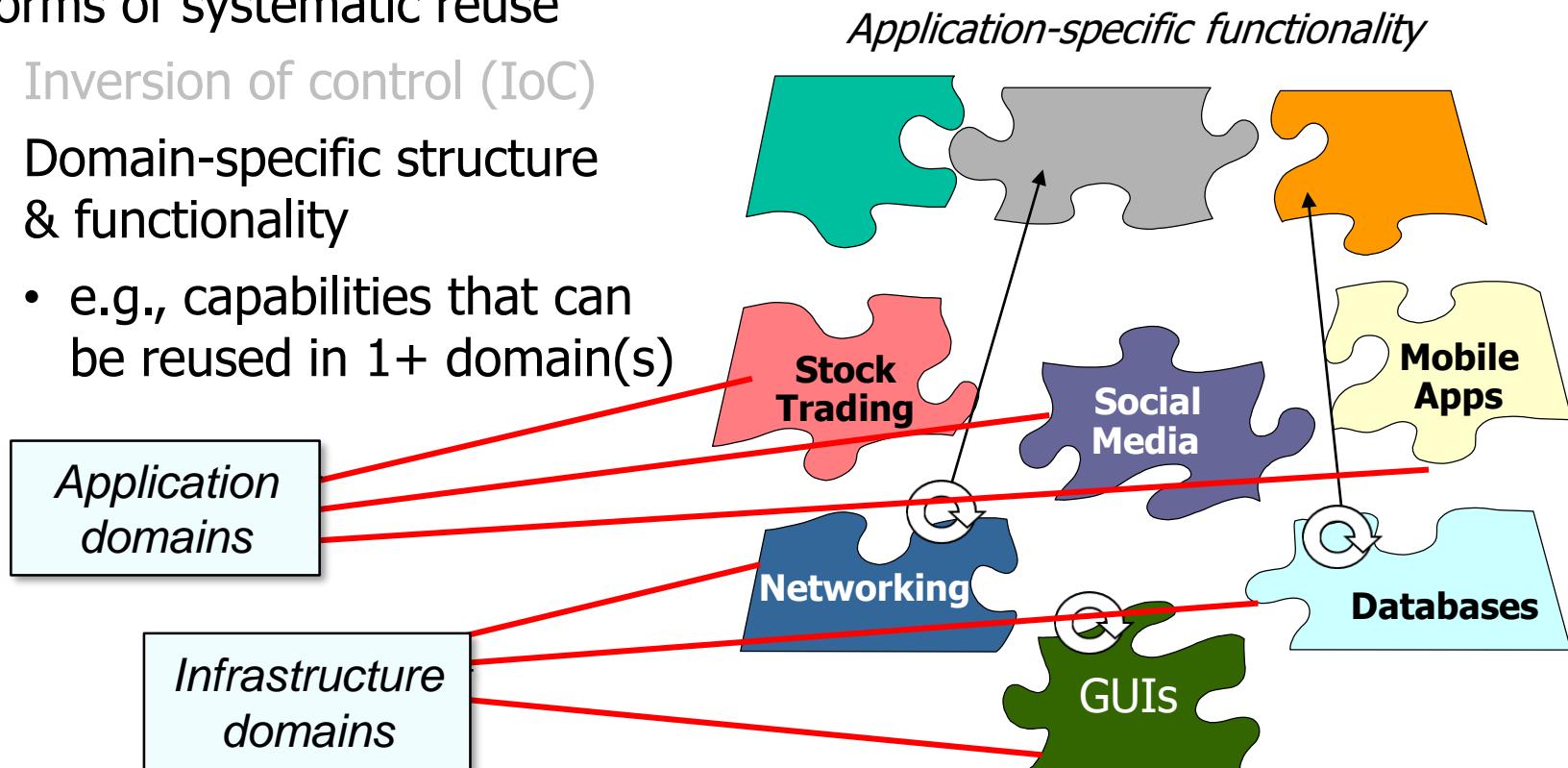
- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Domain-specific structure & functionality



See en.wikipedia.org/wiki/Domain-driven_design

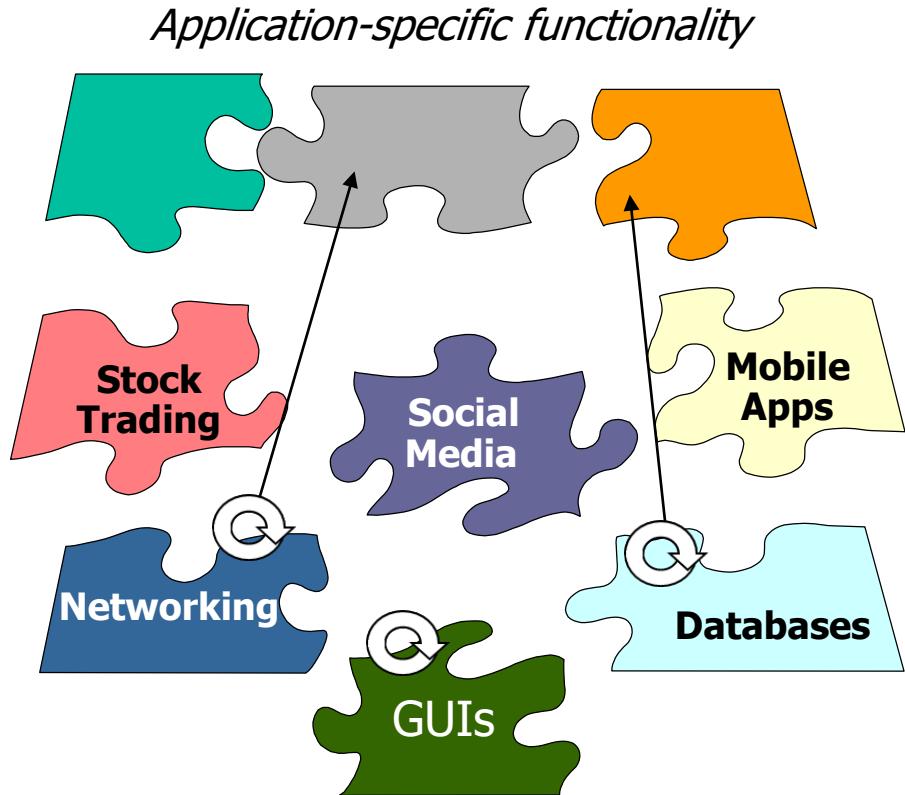
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Domain-specific structure & functionality
 - e.g., capabilities that can be reused in 1+ domain(s)



OO Design of Expression Tree Processing App

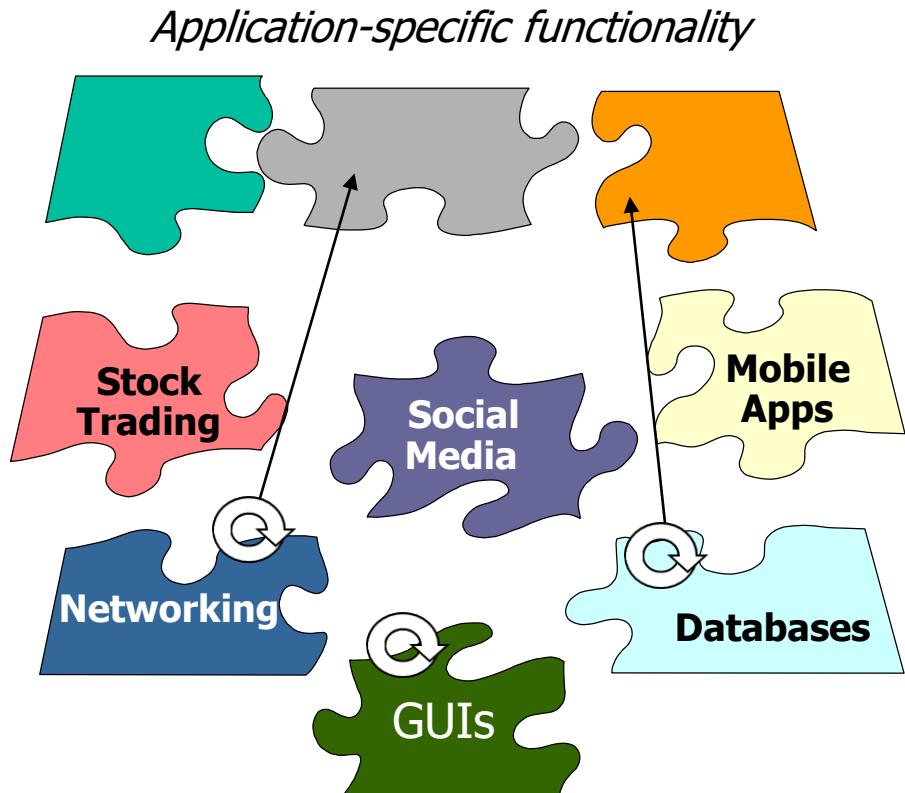
- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Domain-specific structure & functionality
 - e.g., capabilities that can be reused in 1+ domain(s)



Application-specific functionality can systematically reuse framework components

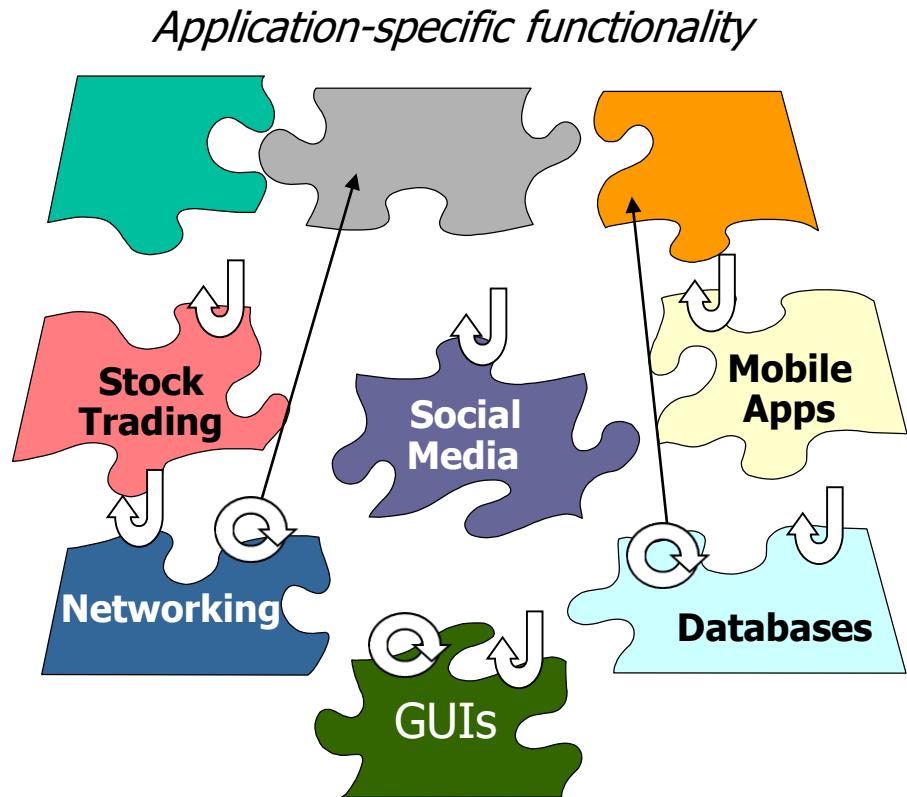
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Domain-specific structure & functionality
 - Semi-complete applications



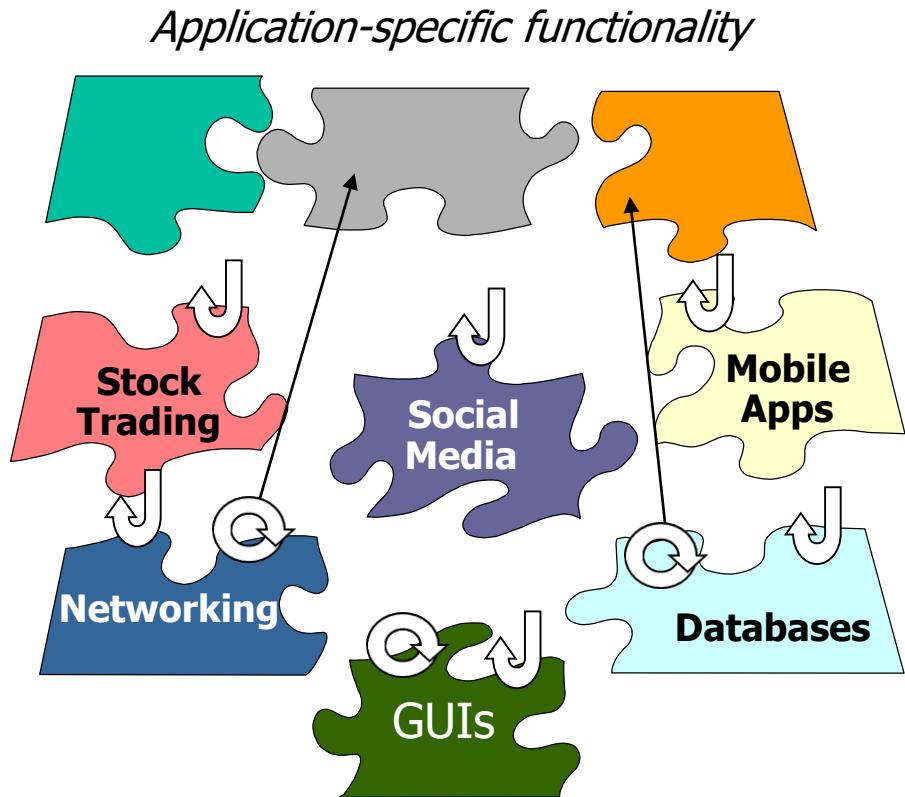
OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Domain-specific structure & functionality
 - Semi-complete applications
 - *Hook methods* plug app logic into the framework



OO Design of Expression Tree Processing App

- Apply “Gang of Four” (GoF) patterns to guide the development of a framework of extensible classes
 - A *framework* is an integrated set of software components that collaborate to provide a reusable architecture for a family of related applications
 - Frameworks exhibit three characteristics that differentiate them from other forms of systematic reuse
 - Inversion of control (IoC)
 - Domain-specific structure & functionality
 - Semi-complete applications
 - *Hook methods* plug app logic into the framework
 - Mediate interactions among *common abstract* & *variant* concrete classes/interfaces



e.g., Java Runnable is an abstract interface providing basis for concrete variants

OO Design of Expression Tree Processing App

- Integrate pattern-oriented language & library features with frameworks
 - Both an app-specific framework...

```
ExpressionTree tree = ...;  
Visitor printVisitor = ...;
```

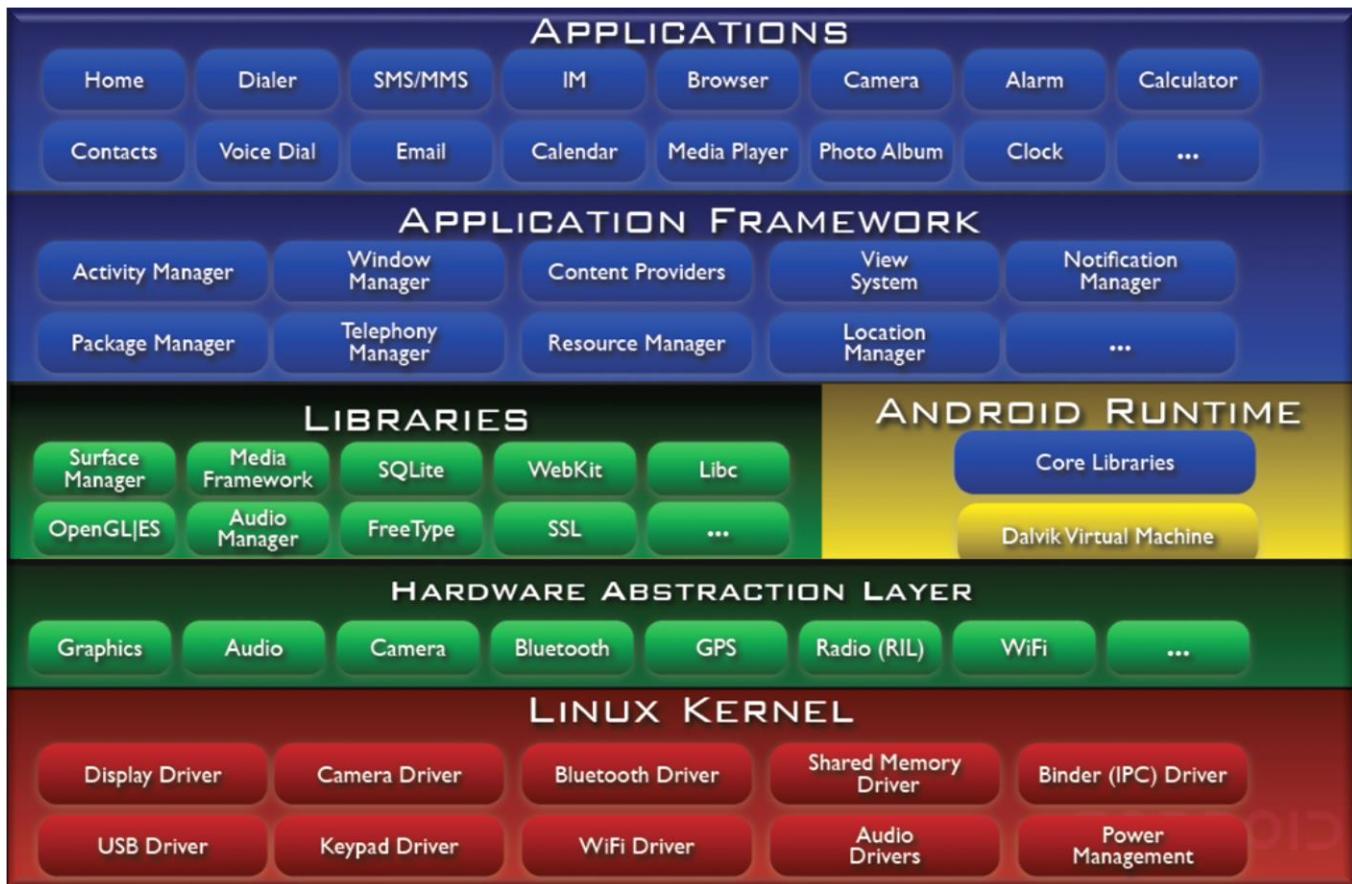
```
for(Iterator<ExpressionTree> iter =  
    tree.iterator(traversalOrder) ;  
    iter.hasNext() ;)  
    iter.next().accept(printVisitor);
```



Factory Method, Bridge, Iterator, Strategy, & Visitor patterns

OO Design of Expression Tree Processing App

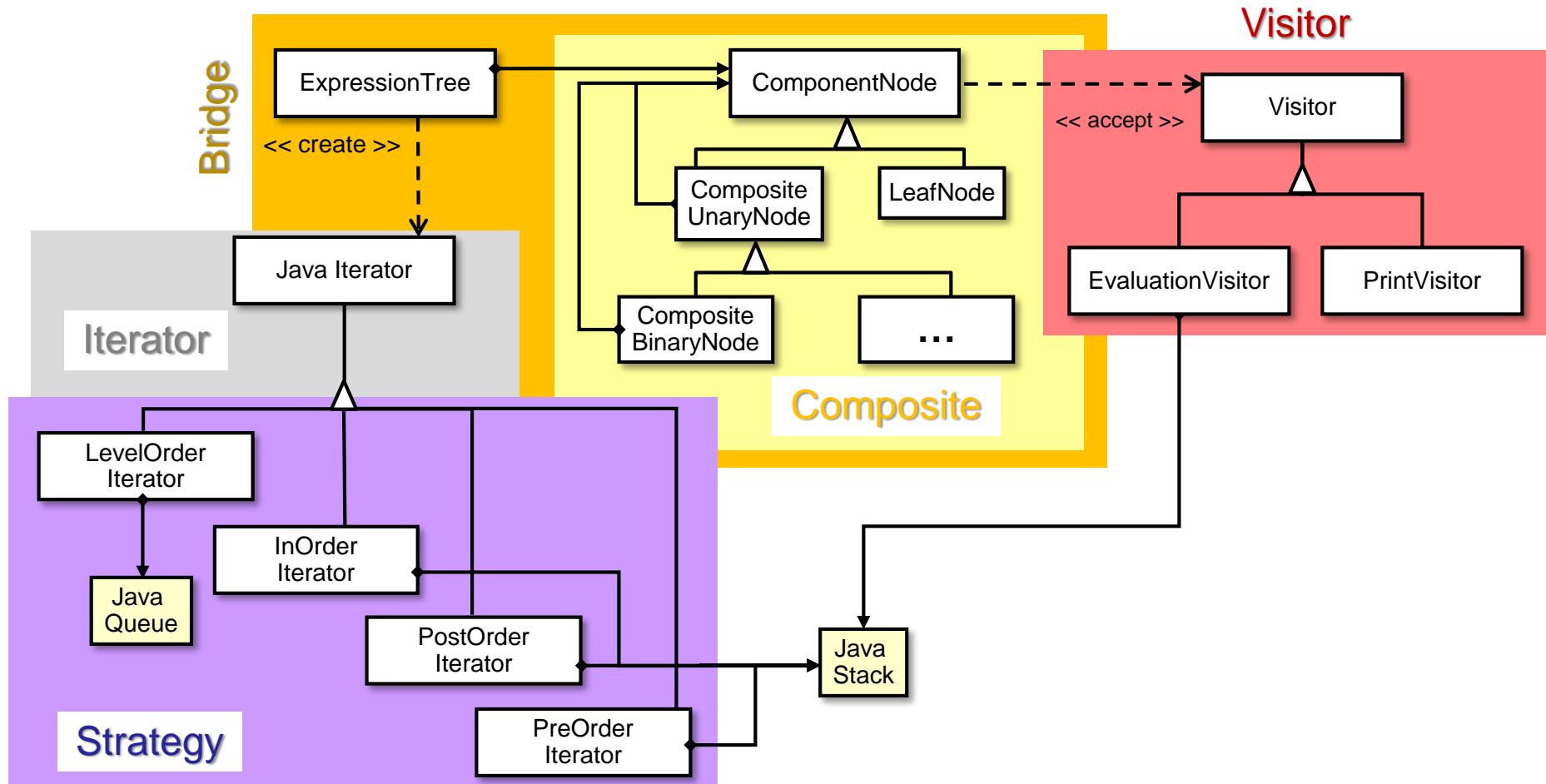
- Integrate pattern-oriented language & library features with frameworks
 - ... & off-the-shelf frameworks...



See developer.android.com for more info on Android

OO Design of Expression Tree Processing App

- Complexity resides in (stable) structure & APIs, rather than (variable) algorithms

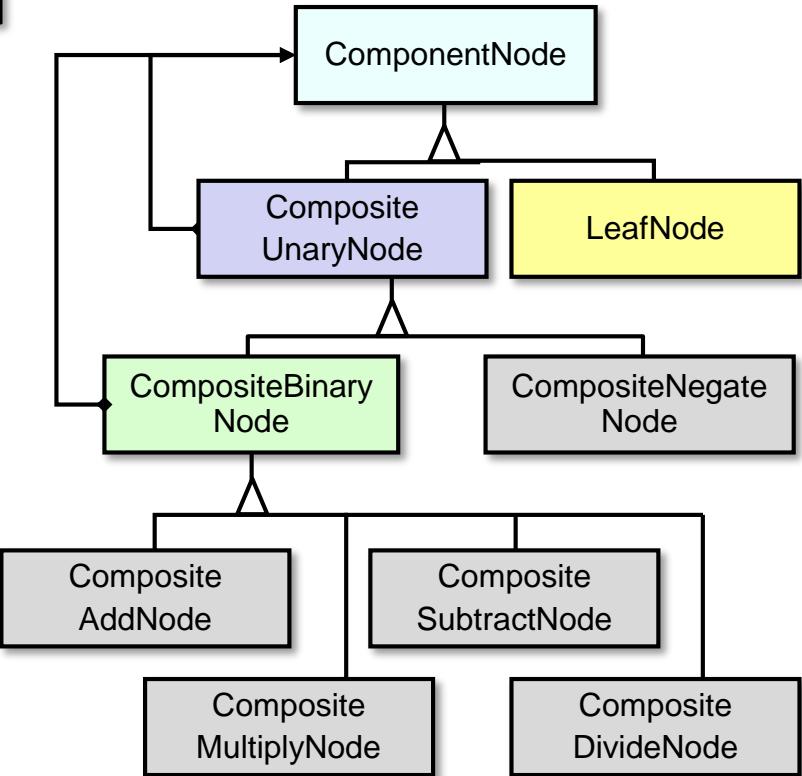
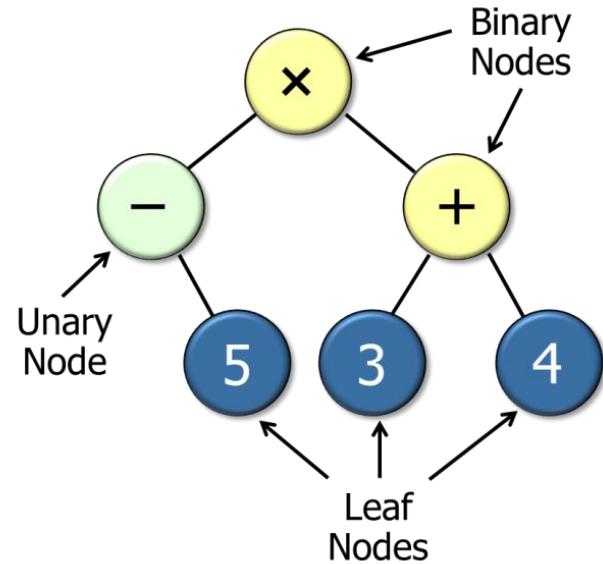


Benefits of Object-Oriented Design

Benefits of Object-Oriented Design

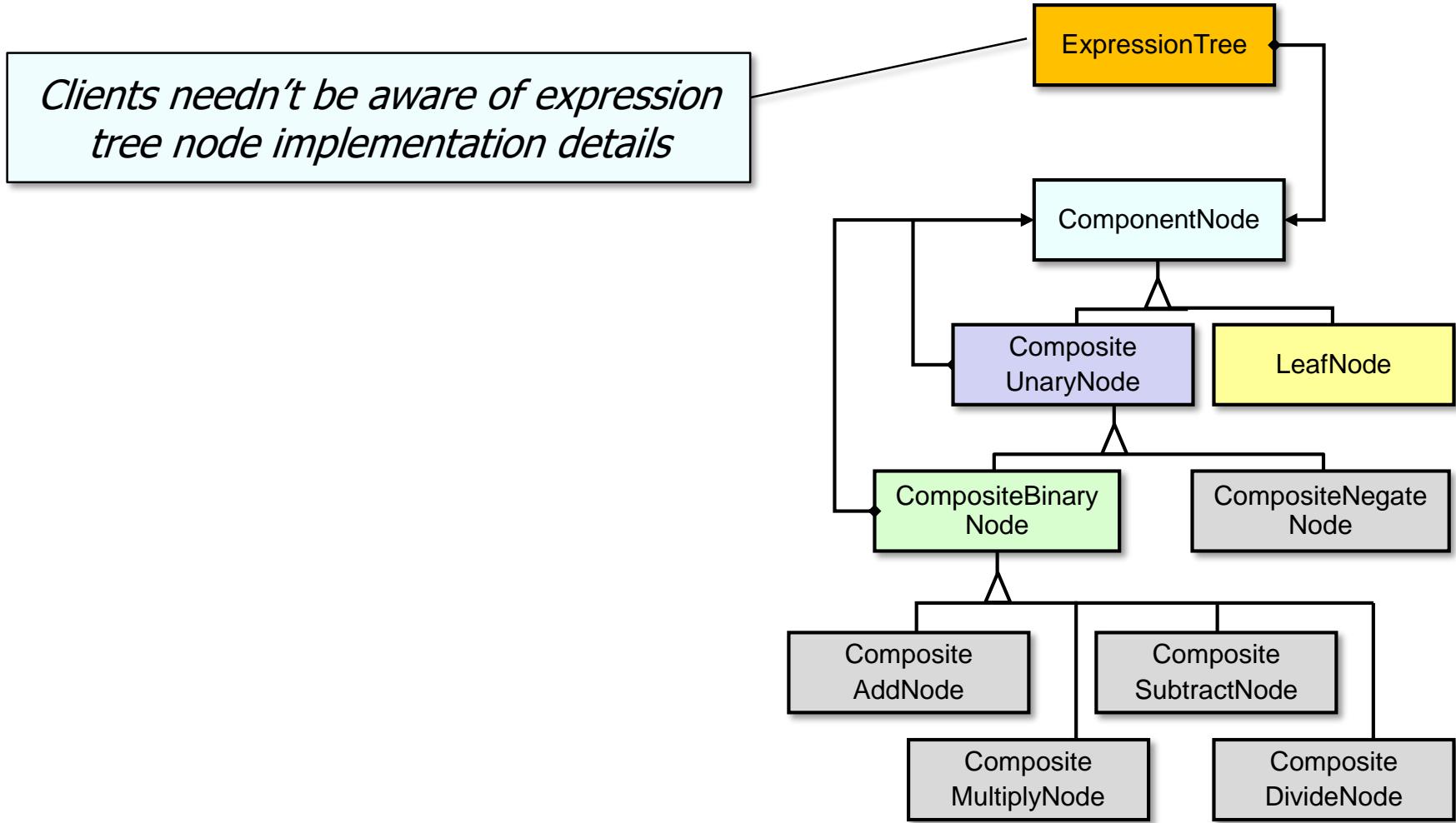
- More accurate modeling of the application domain

Each node in the tree only contains state that's associated with its own capabilities



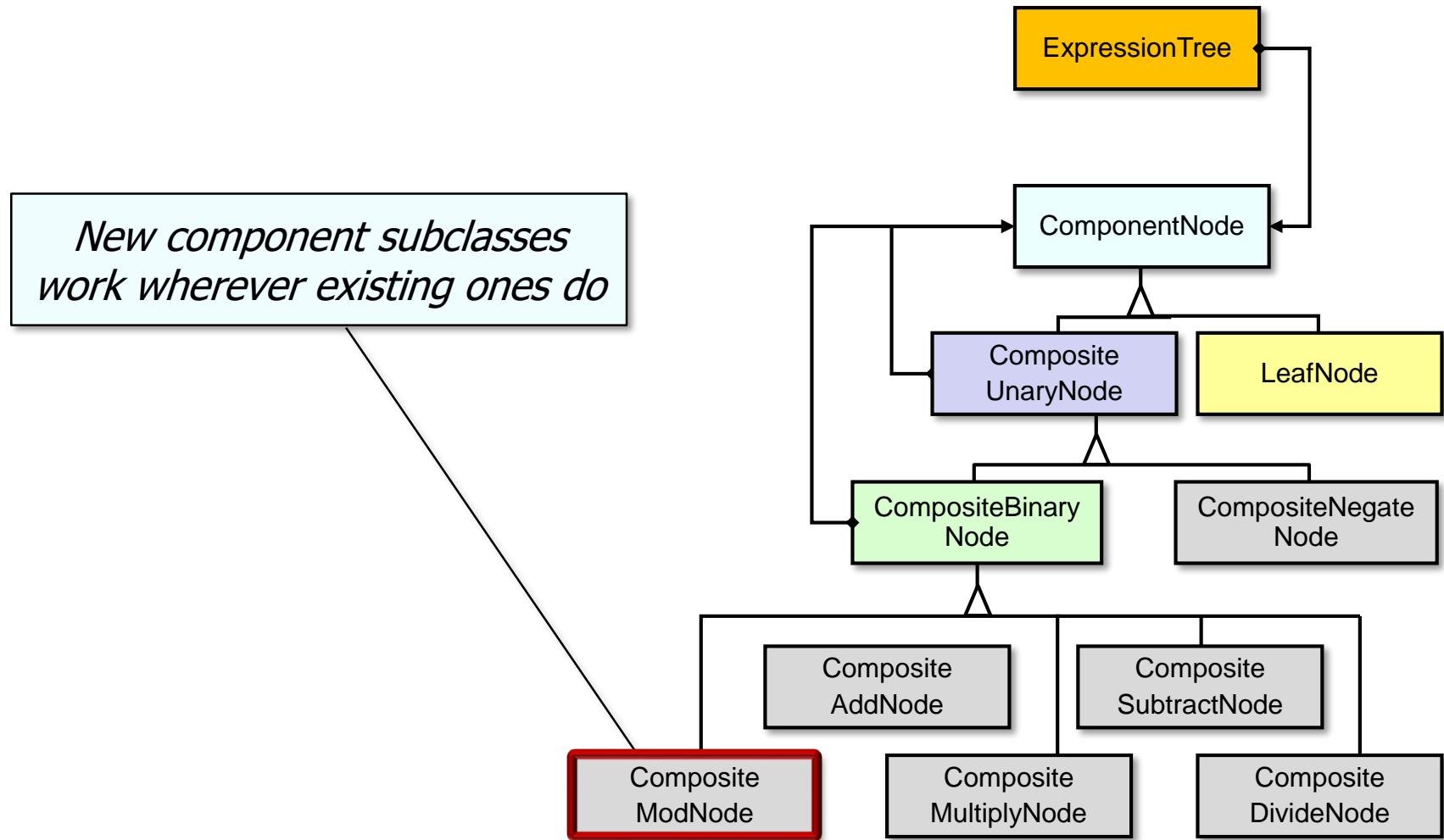
Benefits of Object-Oriented Design

- More effective encapsulation



Benefits of Object-Oriented Design

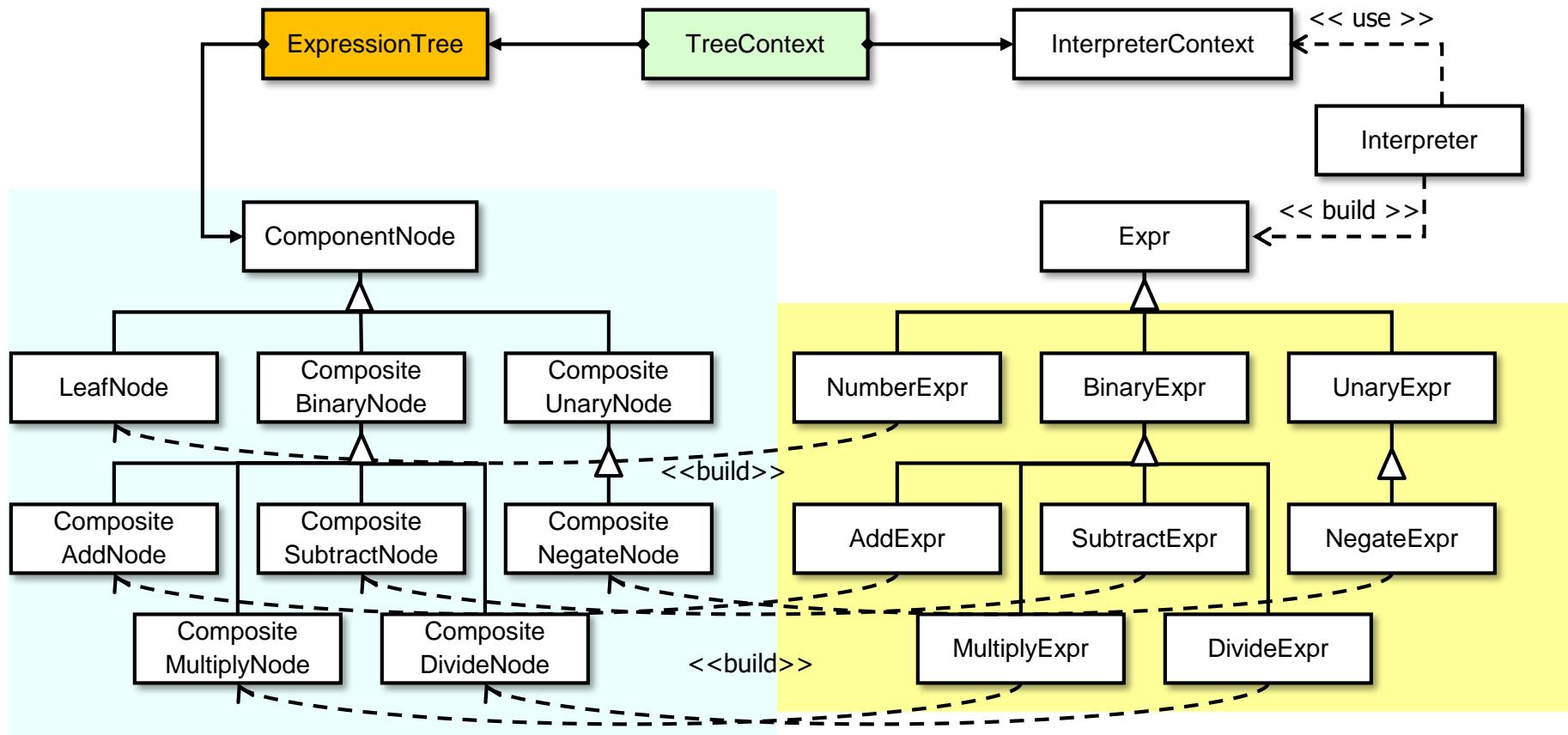
- Straightforward to extend the app to add new types of nodes



Limitations with Object-Oriented Design

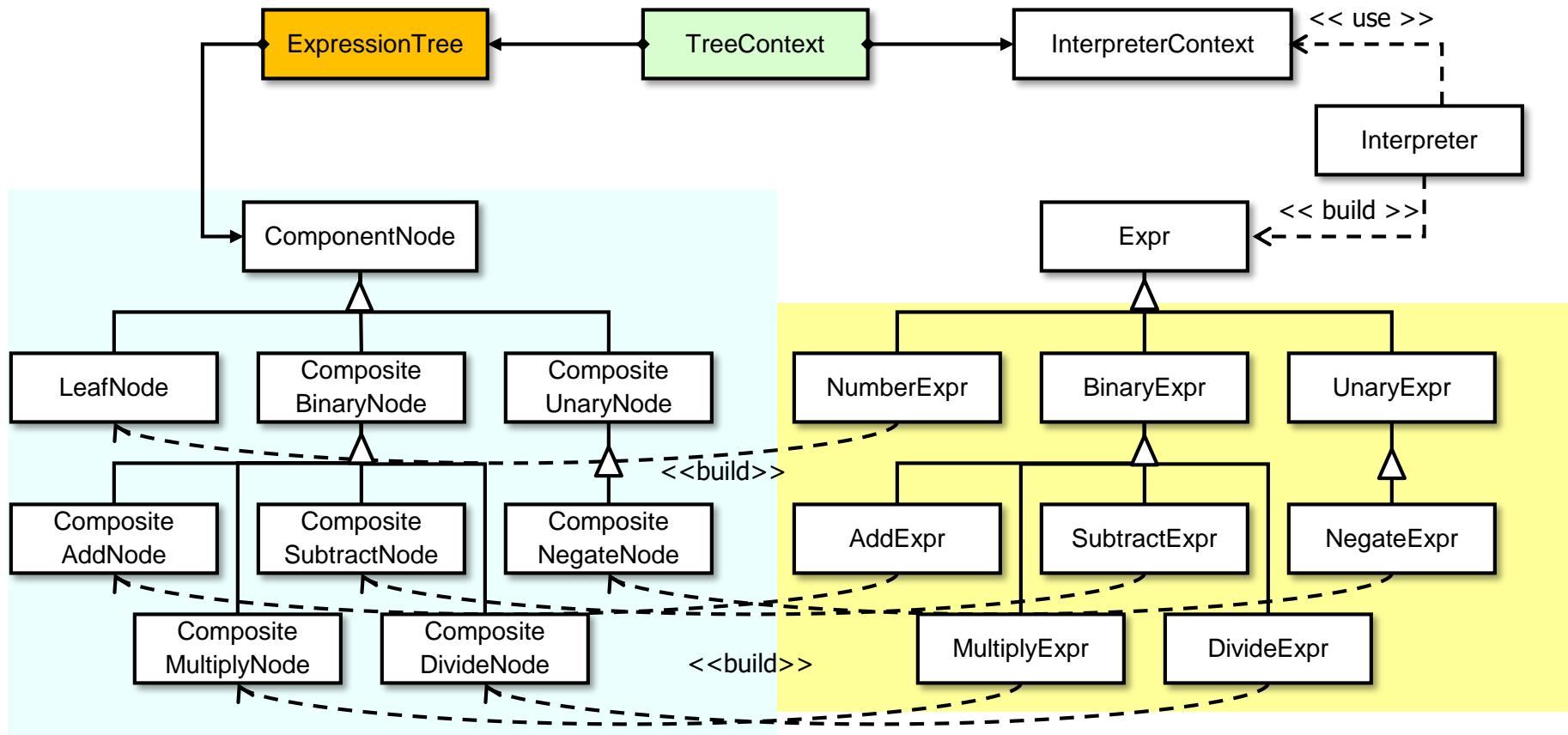
Limitations with Object-Oriented Design

- Solution may be overly rich in classes & associated structures

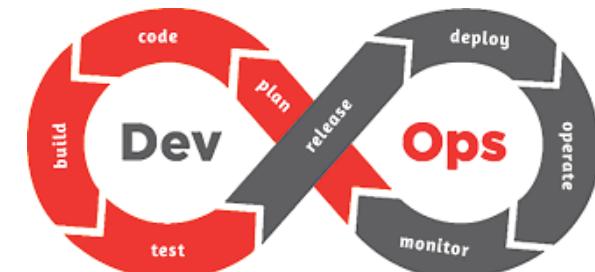


Limitations with Object-Oriented Design

- Solution may be overly rich in classes & associated structures



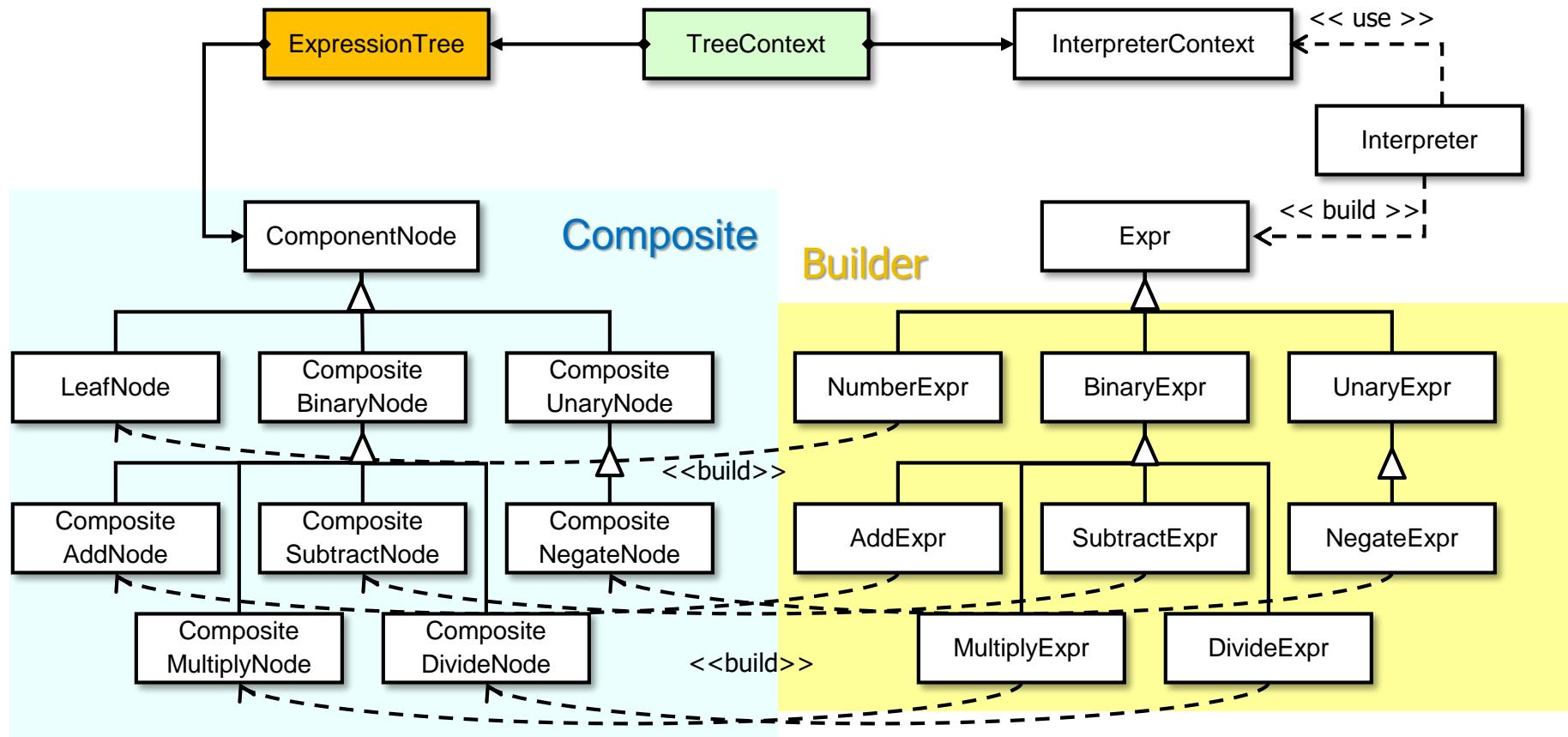
- Good configuration management is needed to handle many files & dependencies



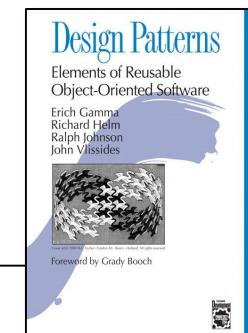
See en.wikipedia.org/wiki/Configuration_management

Limitations with Object-Oriented Design

- Solution may be overly rich in classes & associated structures

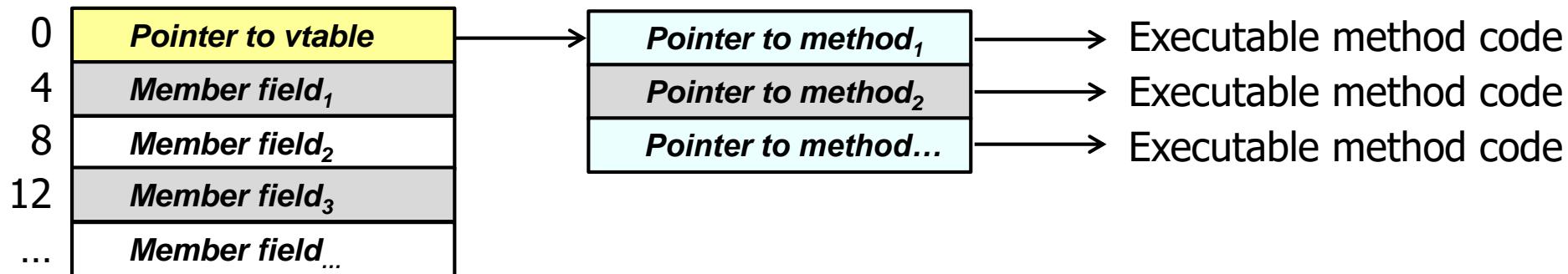


- Good configuration management is needed to handle many files & dependencies
- Knowledge of patterns is also essential!!!!



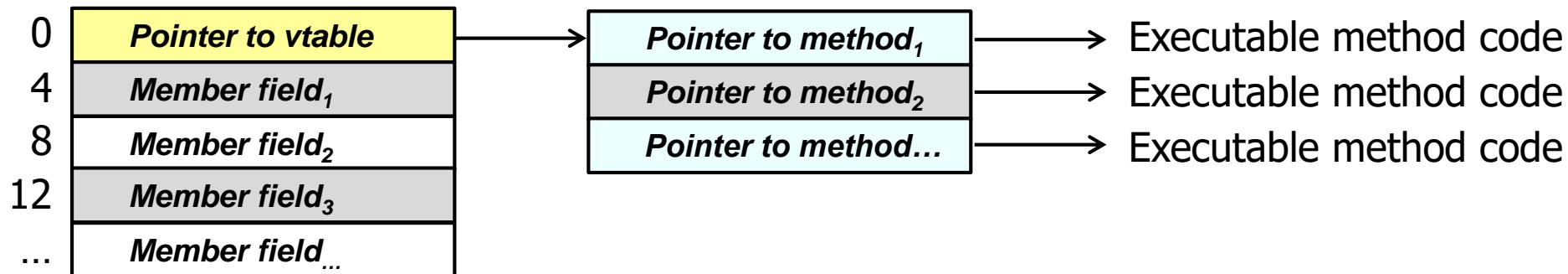
Limitations with Object-Oriented Design

- May be somewhat less efficient than original algorithm decomposition due to time/space overhead of virtual method dispatching



Limitations with Object-Oriented Design

- May be somewhat less efficient than original algorithm decomposition due to time/space overhead of virtual method dispatching



- Modern Java compilers can optimize virtual method dispatching so it's as efficient as large switch statements or if/else chains

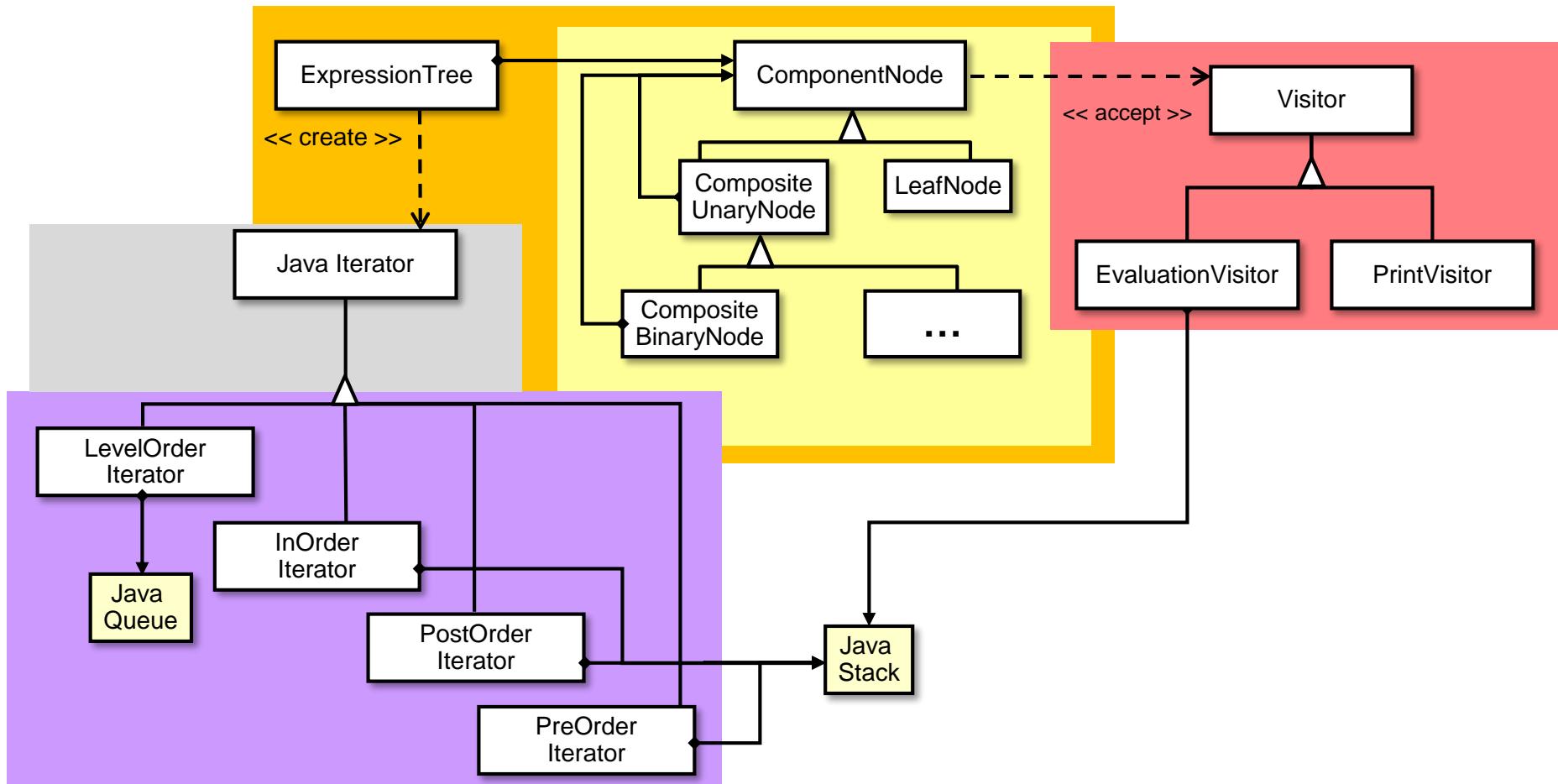


See shipilev.net/blog/2015/black-magic-method-dispatch

Putting All the Pieces Together

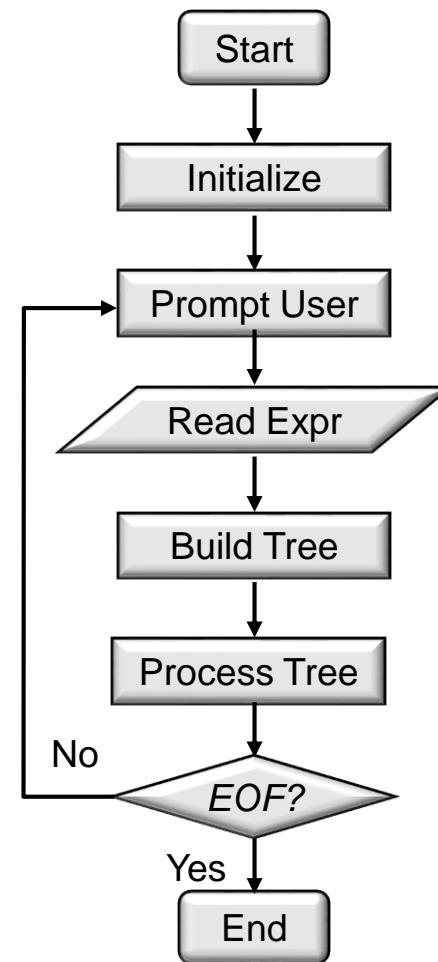
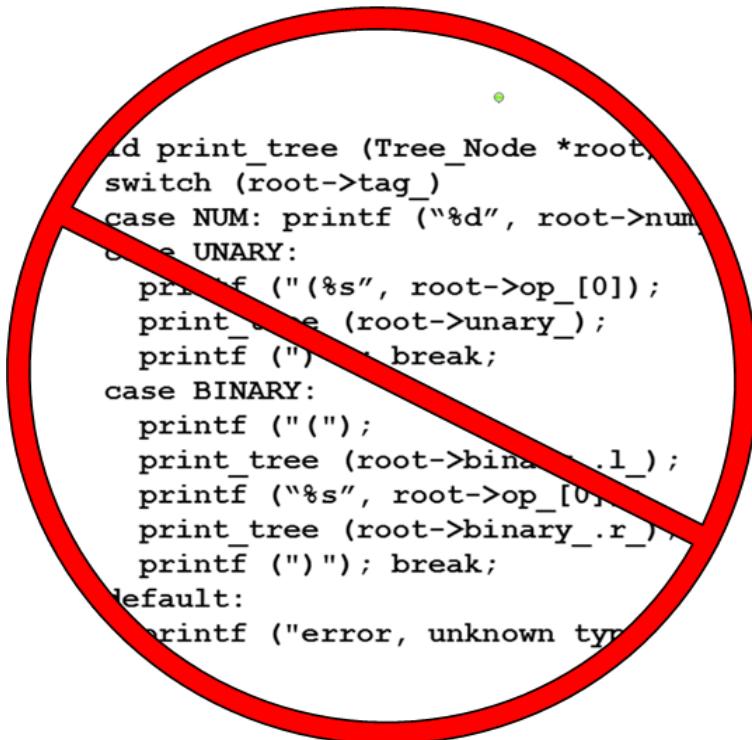
Putting All the Pieces Together

- OO designs are characterized by structuring software architectures around objects & classes in specific domains



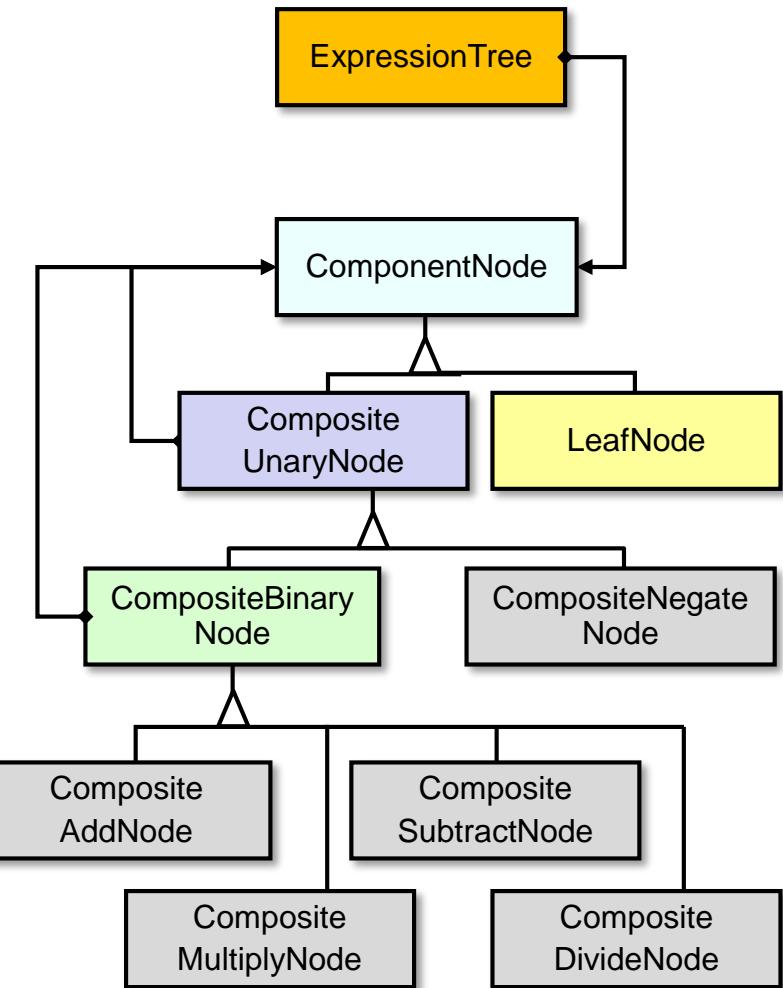
Putting All the Pieces Together

- OO designs are characterized by structuring software architectures around objects & classes in specific domains
 - Rather than on actions performed by the software



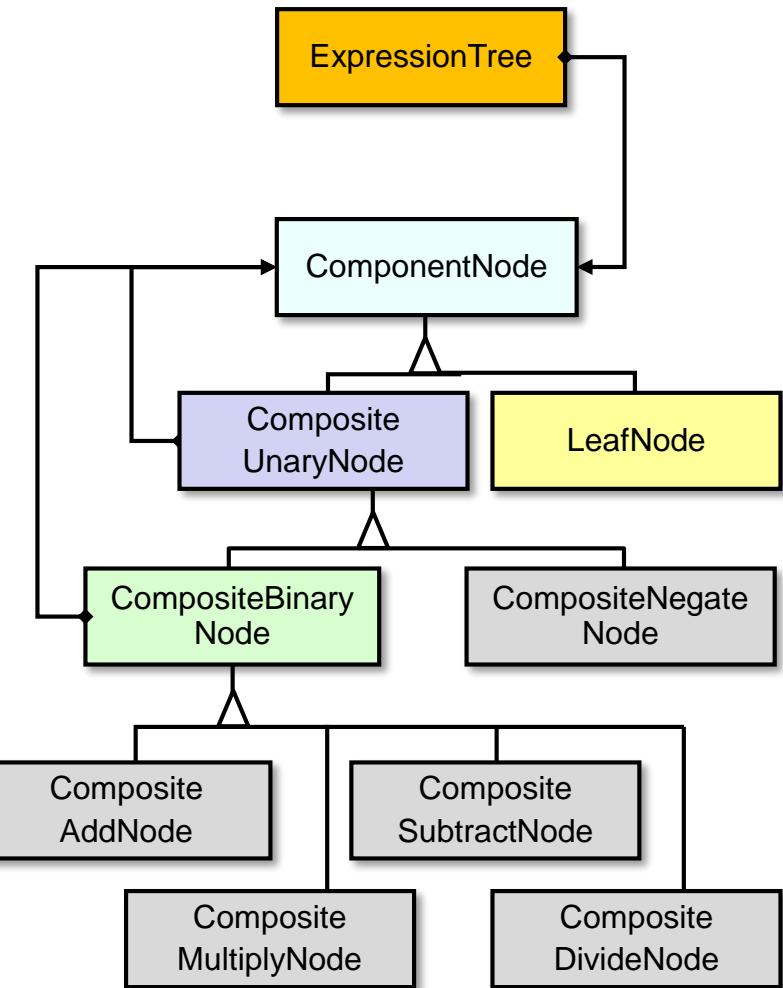
Putting All the Pieces Together

- OO designs are characterized by structuring software architectures around objects & classes in specific domains
- Systems evolve & functionality changes, but well-defined objects & class roles & relationships are often relatively stable over time



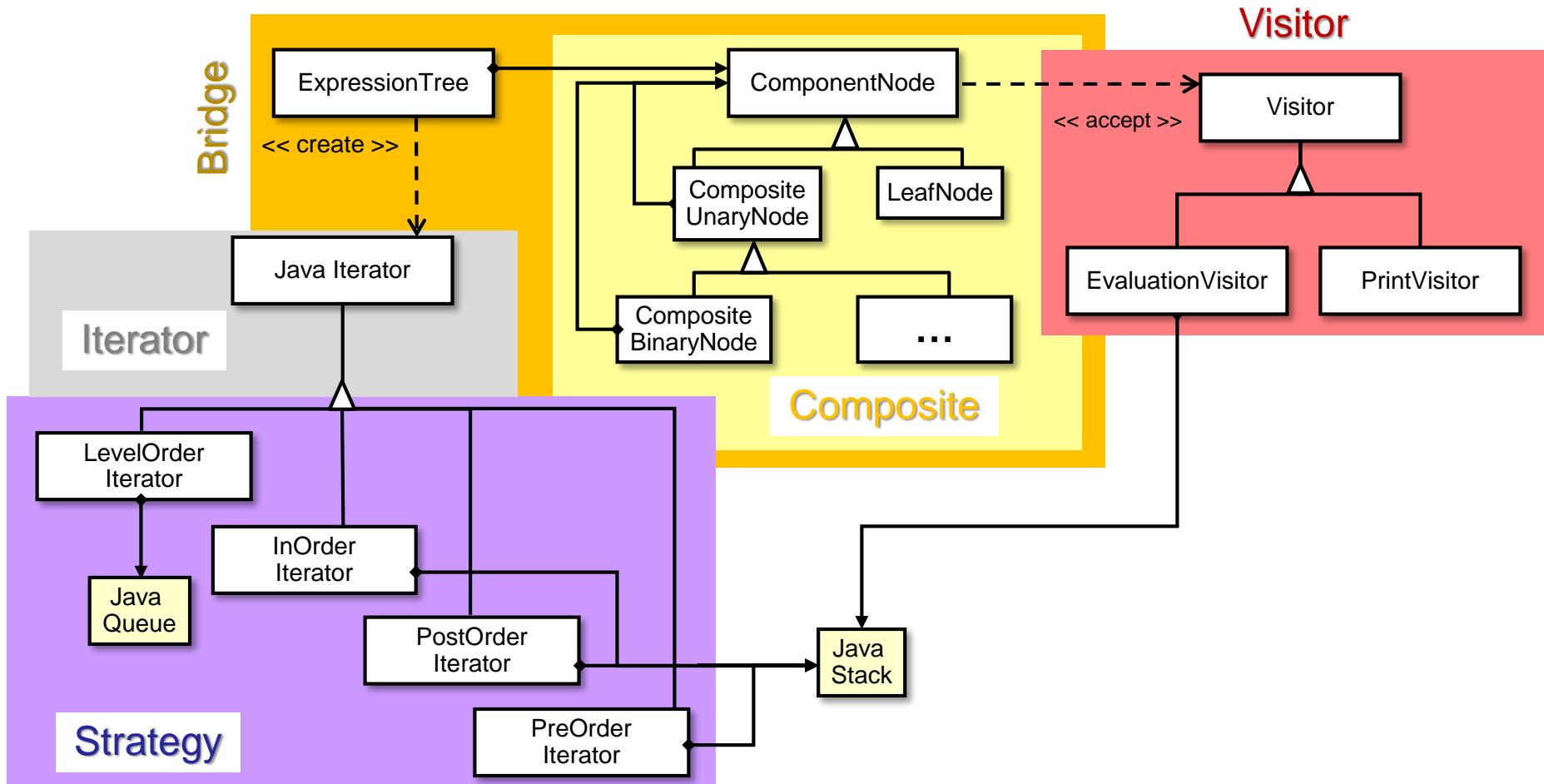
Putting All the Pieces Together

- OO designs are characterized by structuring software architectures around objects & classes in specific domains
- Systems evolve & functionality changes, but well-defined objects & class roles & relationships are often relatively stable over time
- To enhance flexibility & reuse, therefore, it's often better to base the structure of software on the objects & classes rather than on the actions



Putting All the Pieces Together

- Knowledge of patterns is essential to understand key roles & responsibilities in complex software systems



End of Evaluating the Object-Oriented Design of the Expression Tree Processing App

Overview of Patterns Used in the Expression Tree Processing App

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

**Institute for Software
Integrated Systems**

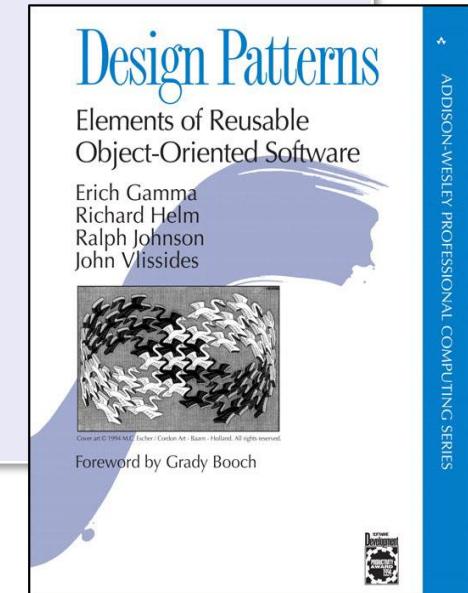
**Vanderbilt University
Nashville, Tennessee, USA**



Lesson Intro

- The book *Design Patterns: Elements of Reusable Object-Oriented Software* (the so-called "Gang of Four" or "GoF" book) presents recurring solutions to common problems in software design in the form of 23 patterns

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



Learning Objectives

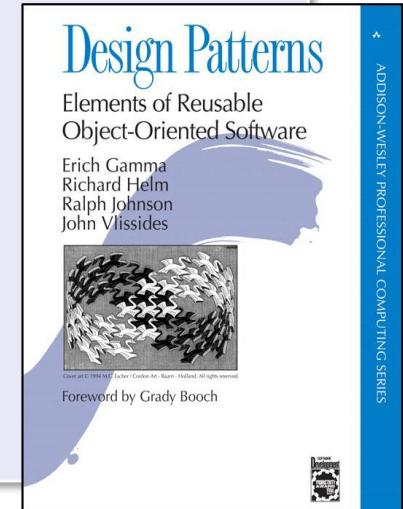
Learning Objective

- Recognize which GoF patterns the expression tree processing app uses

	Creational	Structural	Behavioral
Class	Factory Method ✓	Adapter ✓ (class)	Interpreter ✓ Template Method
Object	Abstract Factory ✓ Builder ✓ Prototype Singleton	Adapter ✓ (object) Bridge ✓ Composite ✓ Decorator ✓ Flyweight Façade Proxy	Chain of Responsibility Command ✓ Iterator Mediator Memento Observer State Strategy Visitor



Patterns covered in Day 1



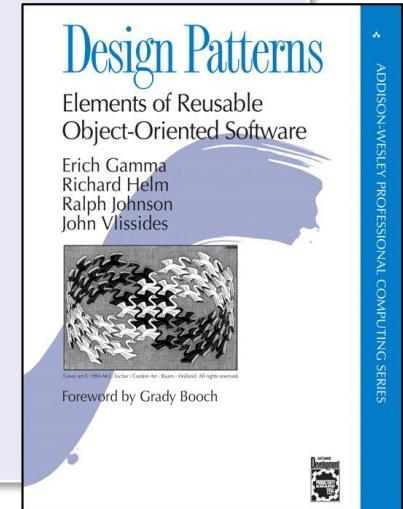
Learning Objective

- Recognize which GoF patterns the expression tree processing app uses

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method ✓
Object	Abstract Factory Builder Prototype Singleton ✓	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator ✓ Mediator Memento Observer ✓ State ✓ Strategy ✓ Visitor ✓



Patterns covered in Day 2



Design Problems & GoF Pattern Solutions

Design Problems & GoF Pattern Solutions

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

These patterns constitute a "pattern sequence" for the case study app

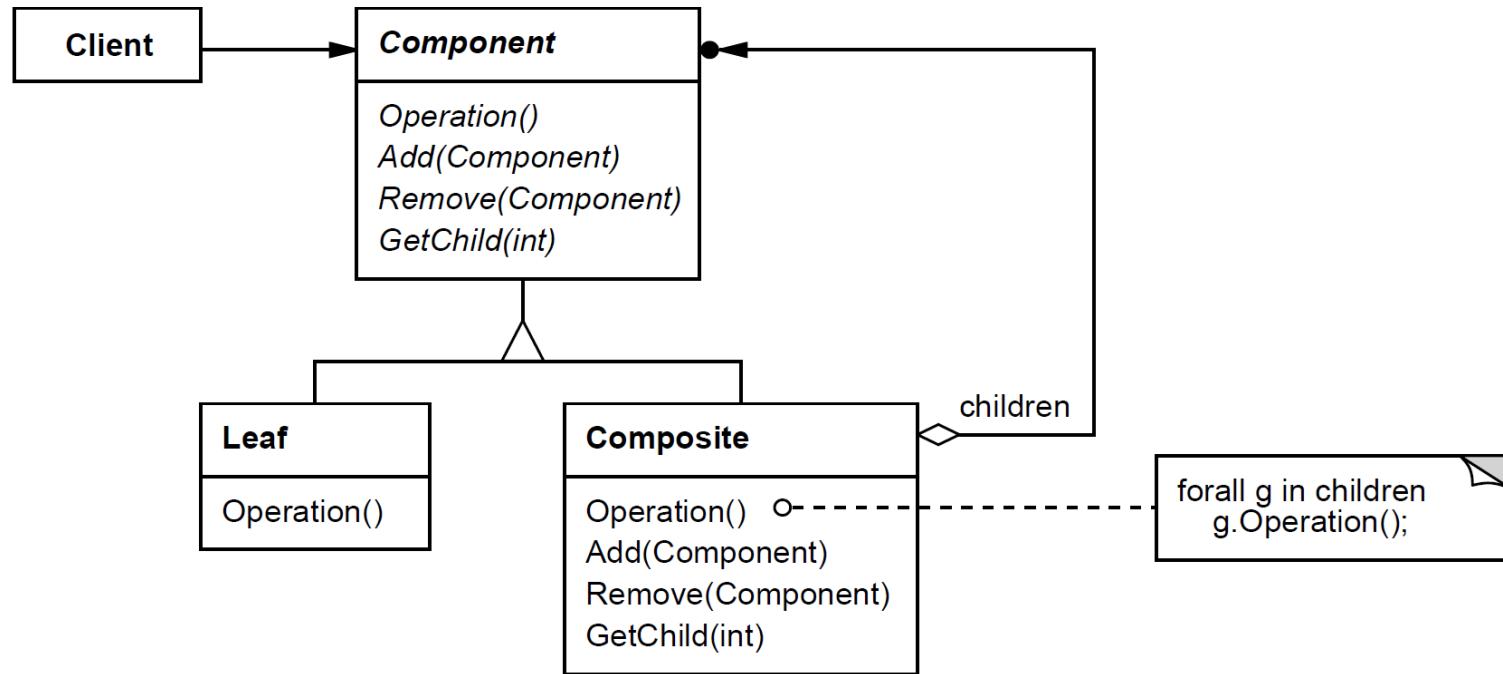
See www.dre.vanderbilt.edu/~schmidt/POSA-tutorial.pdf

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Non-extensible & error-prone designs	Composite

Composite intent

- Treat individual objects & multiple, recursively-composed objects uniformly



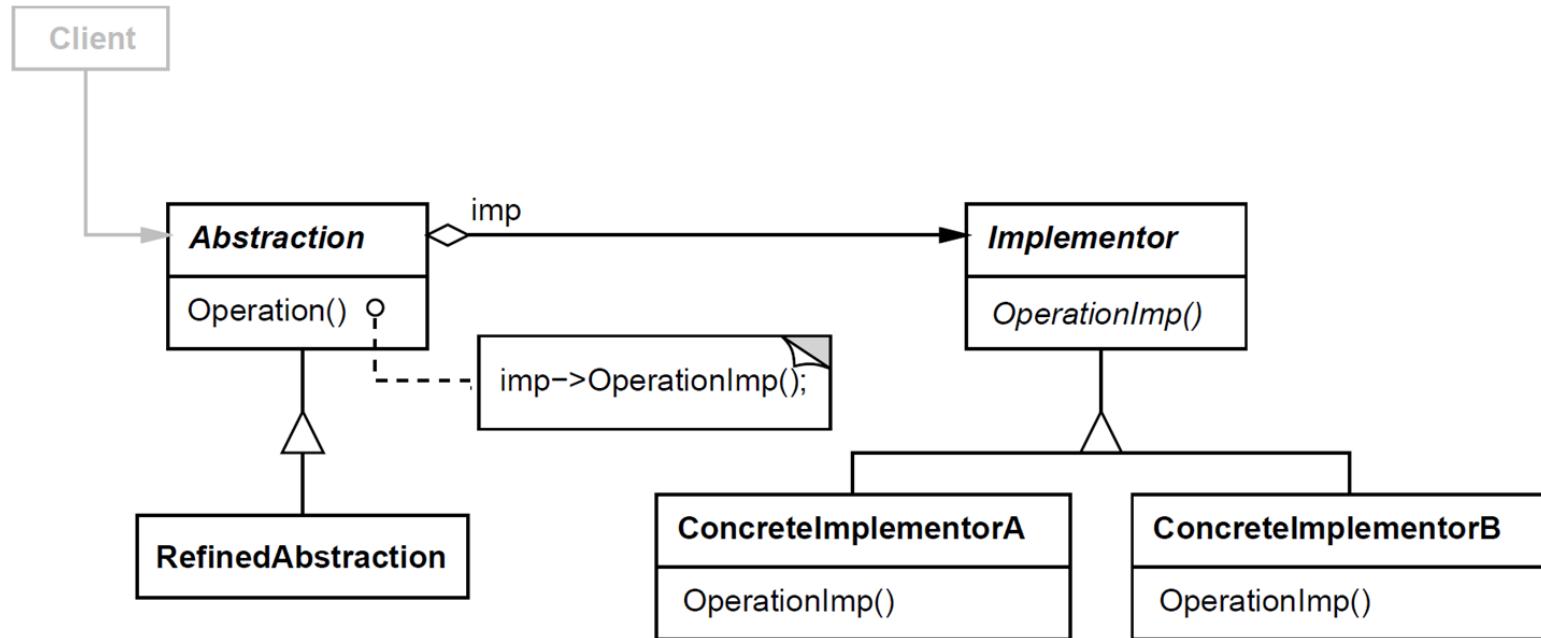
See en.wikipedia.org/wiki/Composite_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Minimizing impact of variability	Bridge

Bridge intent

- Separate an abstraction from its implementation(s) so the two can vary independently



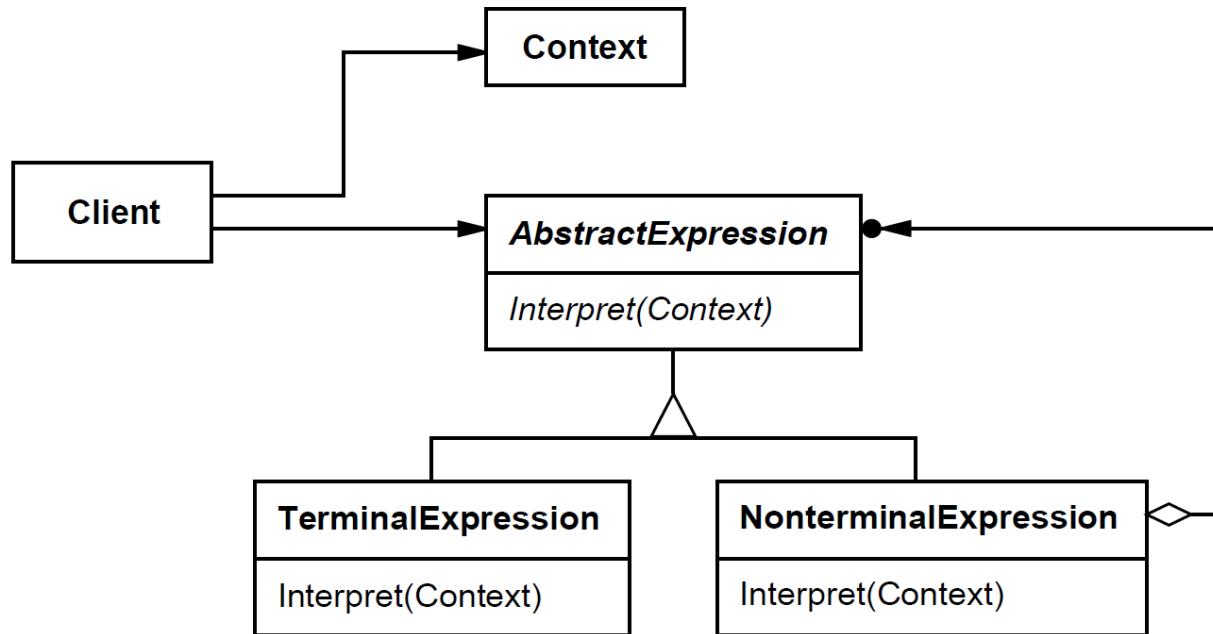
See en.wikipedia.org/wiki/Bridge_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Inflexible expression input processing	Interpreter

Interpreter intent

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

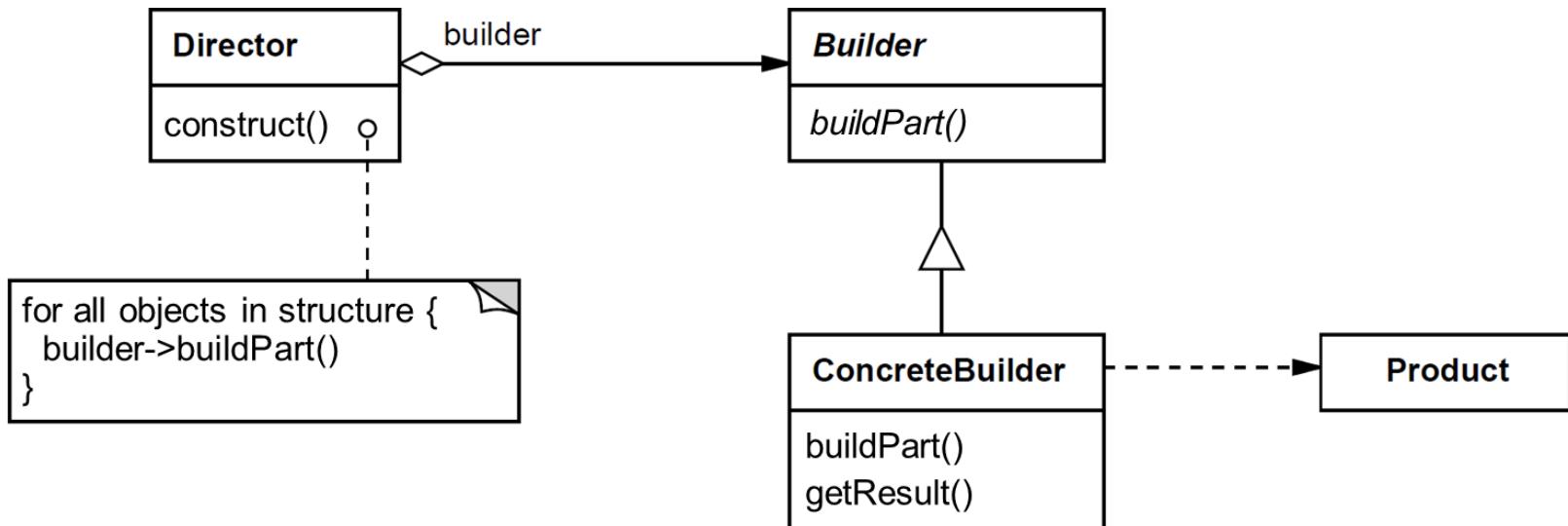


Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Inflexible interpreter output	Builder

Builder intent

- Separate the construction of a complex object from its representation



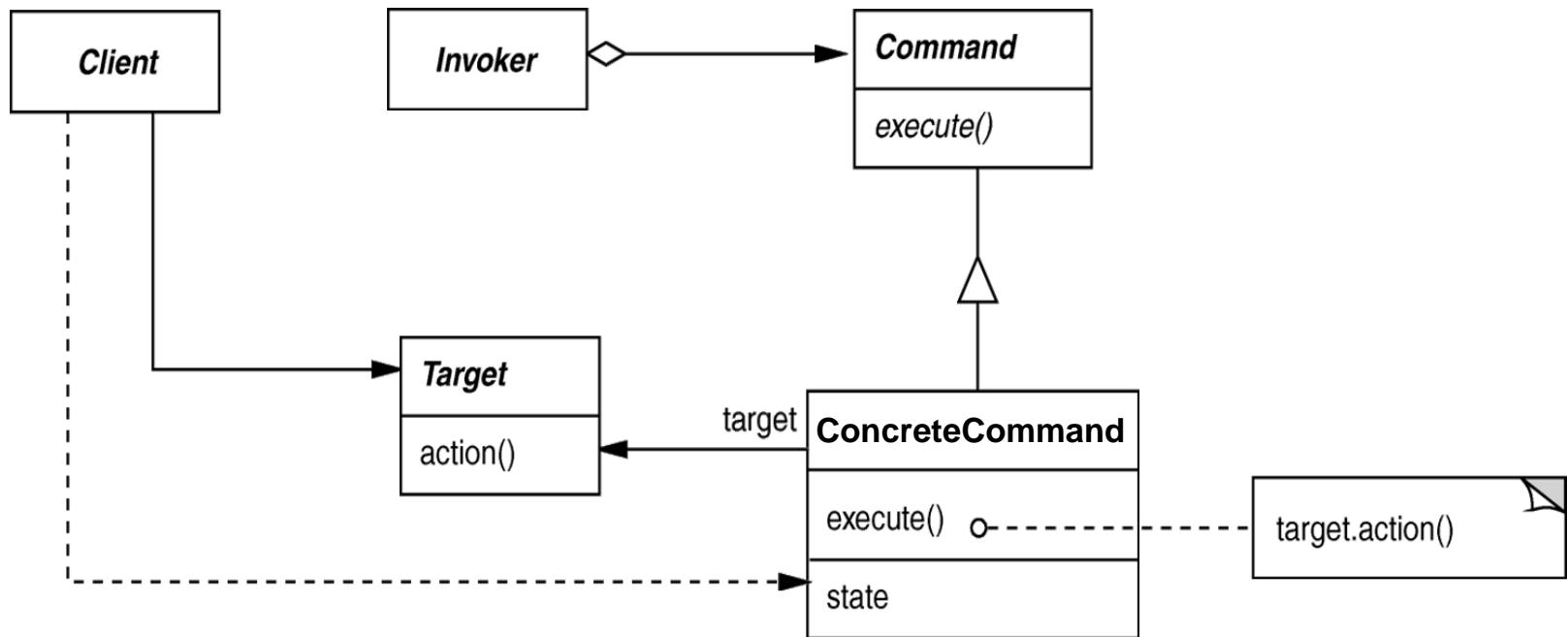
See en.wikipedia.org/wiki/Builder_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Scattered & fixed operation implementations	Command

Command intent

- Encapsulate the request for a service as an object

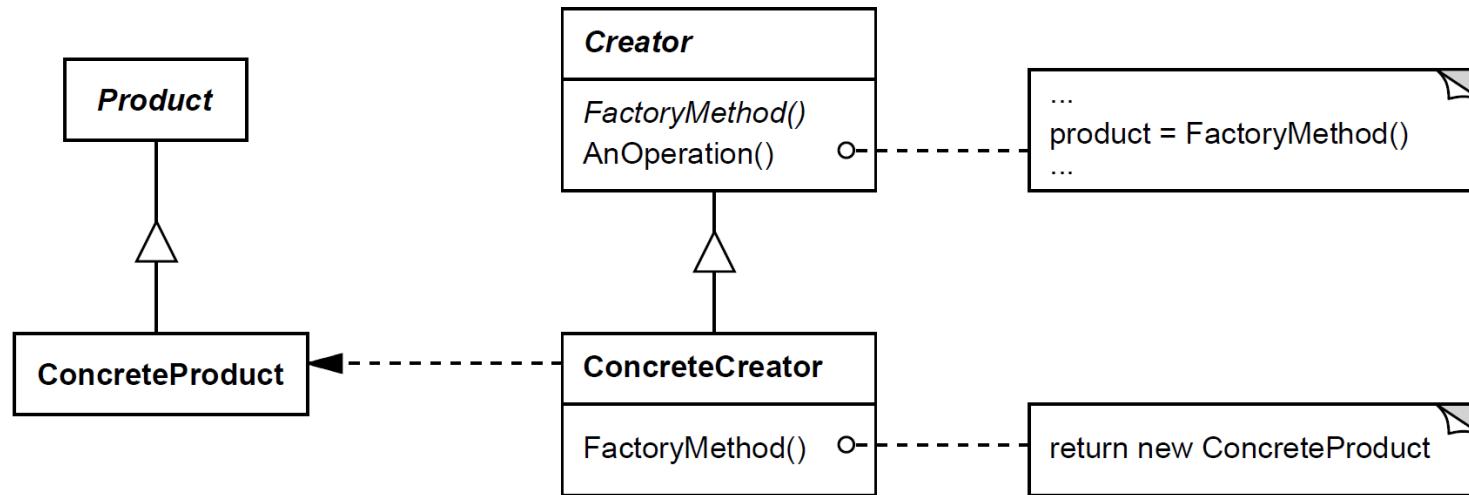


Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Inflexible creation of variabilities	Factory Method

Factory Method intent

- Provide an interface for creating an object, but leave choice of object's concrete type to a subclass



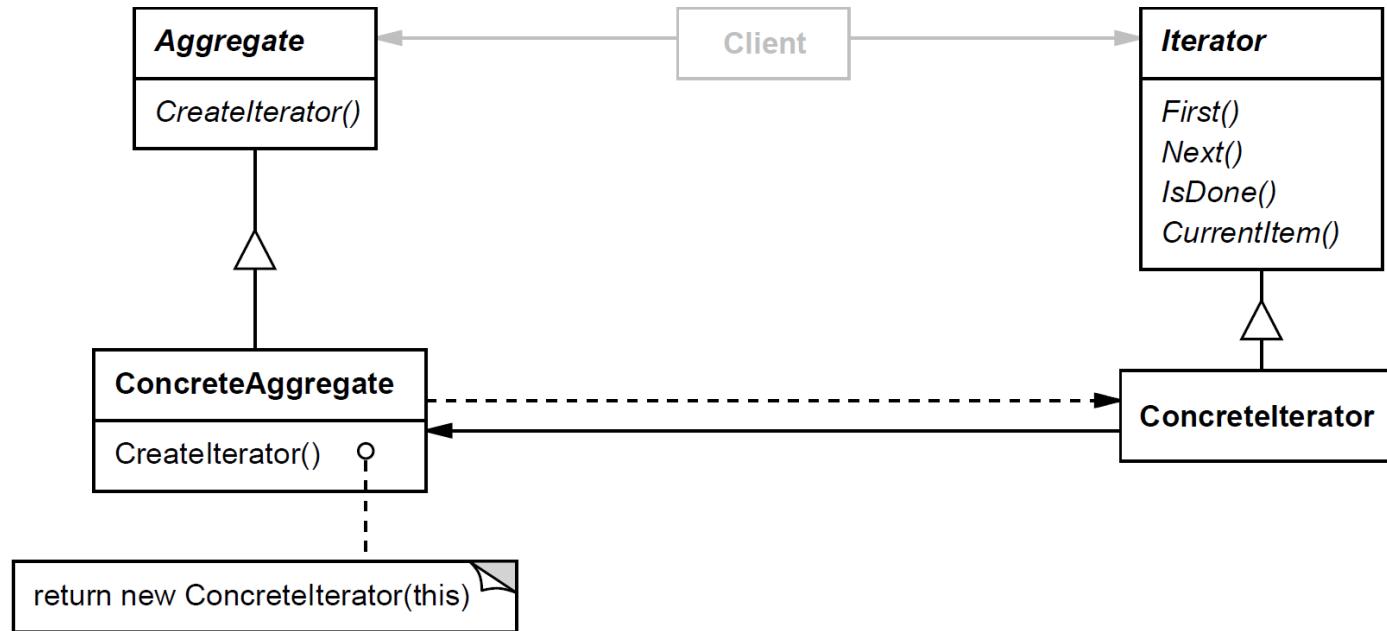
See en.wikipedia.org/wiki/Factory_method_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Inflexible expression tree traversal	Iterator

Iterator intent

- Access elements of an aggregate without exposing its representation



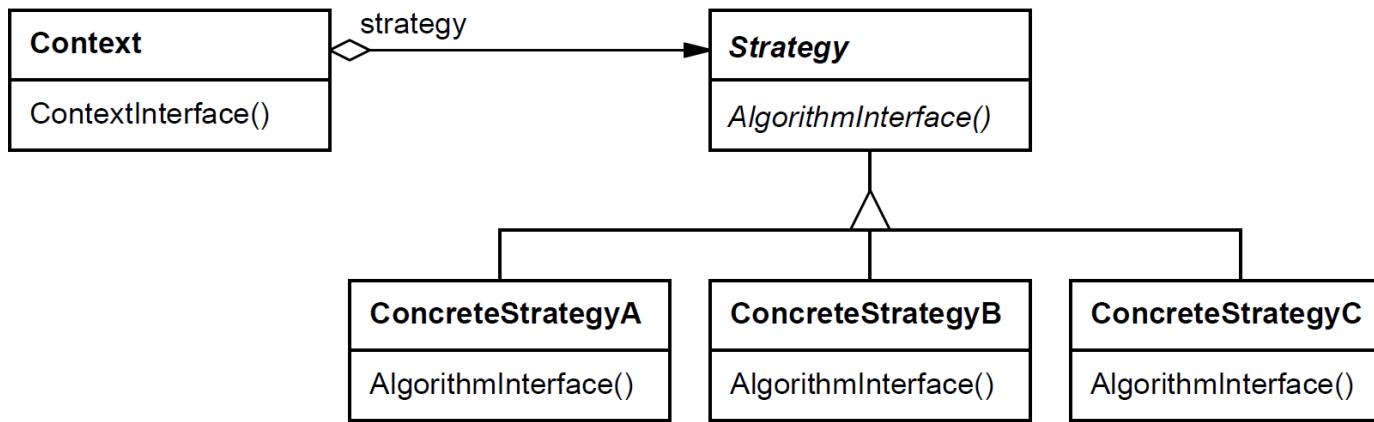
See en.wikipedia.org/wiki/Iterator_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Obtrusive behavior changes	Strategy

Strategy intent

- Define a family of algorithms, encapsulate each one, & make them interchangeable to let clients & algorithms vary independently



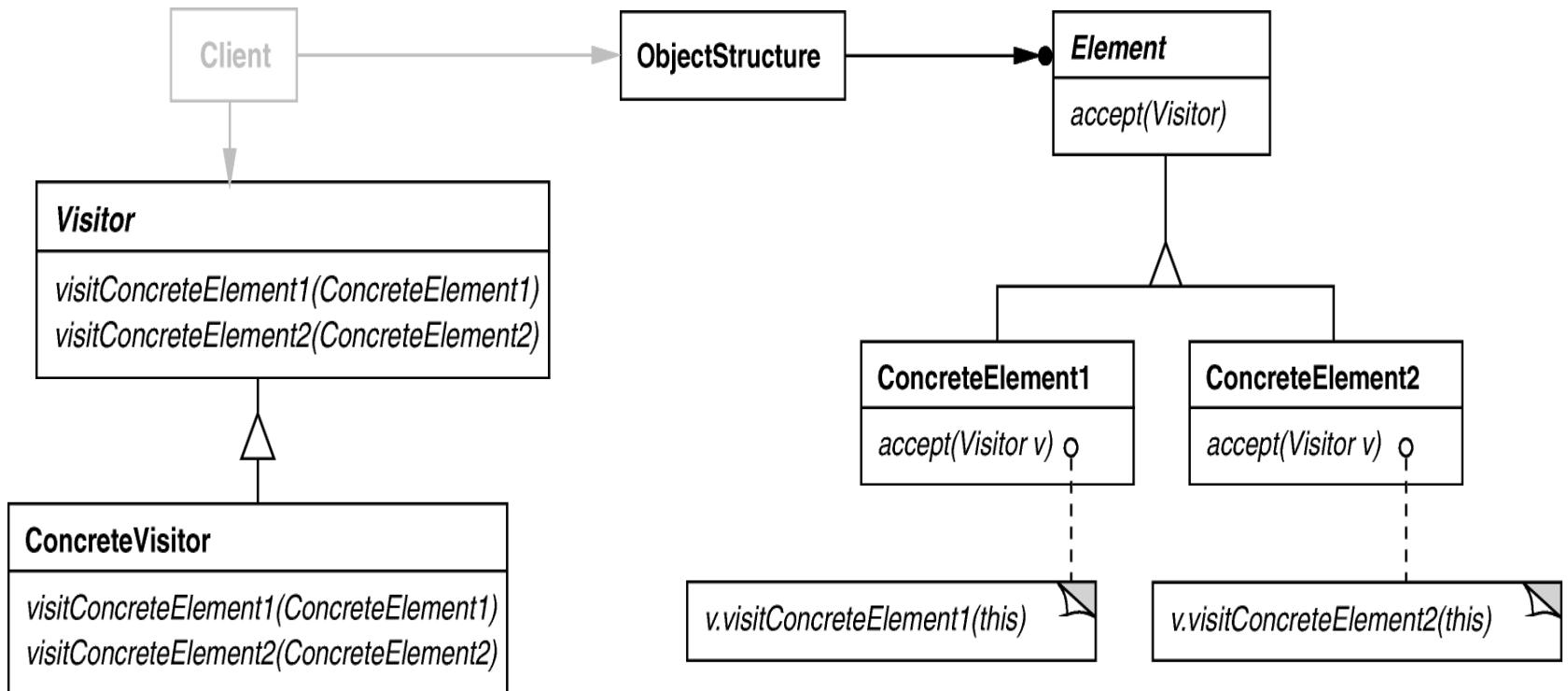
See en.wikipedia.org/wiki/Strategy_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Non-extensible tree operations	Visitor

Visitor intent

- Centralize operations on an object structure so that they can vary independently, but still behave polymorphically



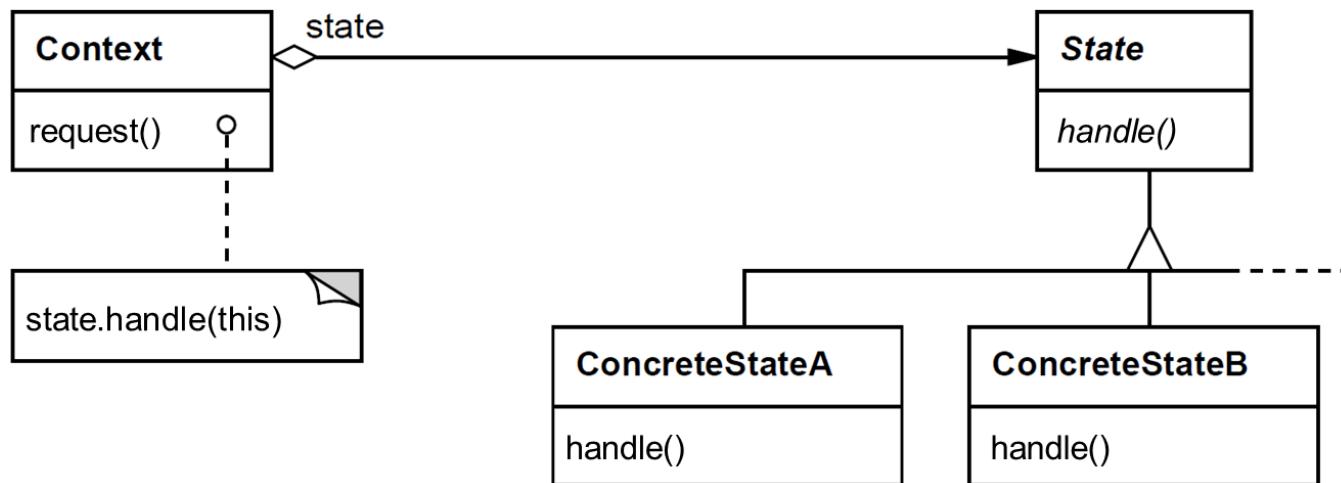
See en.wikipedia.org/wiki/Visitor_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Incorrect operation ordering	State

State intent

- Allow an object to alter its behavior when its internal state changes—the object will appear to change its class



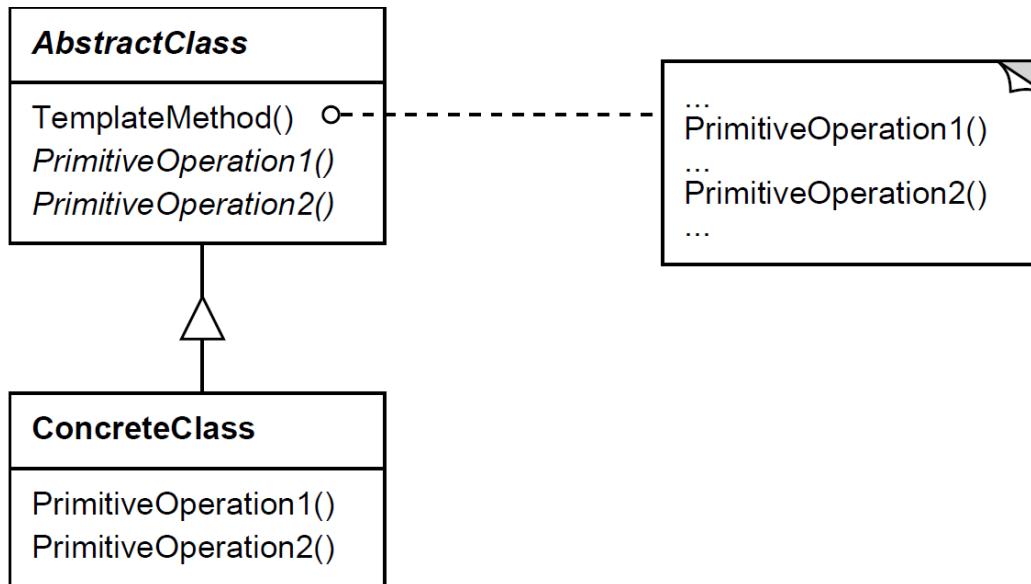
See en.wikipedia.org/wiki/State_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Non-extensible operating modes	Template Method

Template Method intent

- Provide a skeleton of an algorithm in a method, deferring some steps to subclasses



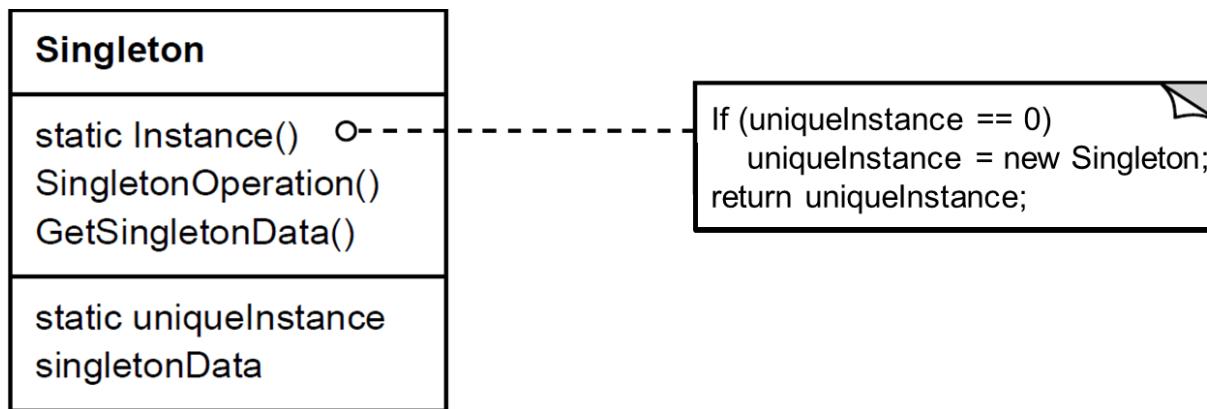
See en.wikipedia.org/wiki/Template_method_pattern

Design Problems & GoF Pattern Solutions

Design Problem	Pattern
Minimizing global variable liabilities	Singleton

Singleton intent

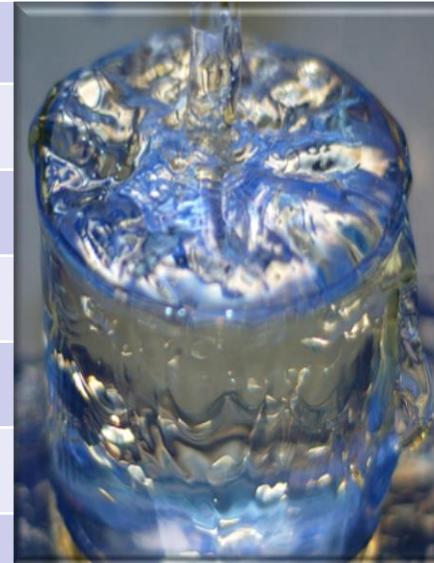
- Ensure a class only has one instance & provide a global point of access



See en.wikipedia.org/wiki/Singleton_pattern

Design Problems & GoF Pattern Solutions

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



Naturally, these patterns apply to more than expression tree processing apps!

End of Overview of Patterns
Used in the Expression Tree
Processing App

The Composite Pattern

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software
Integrated Systems

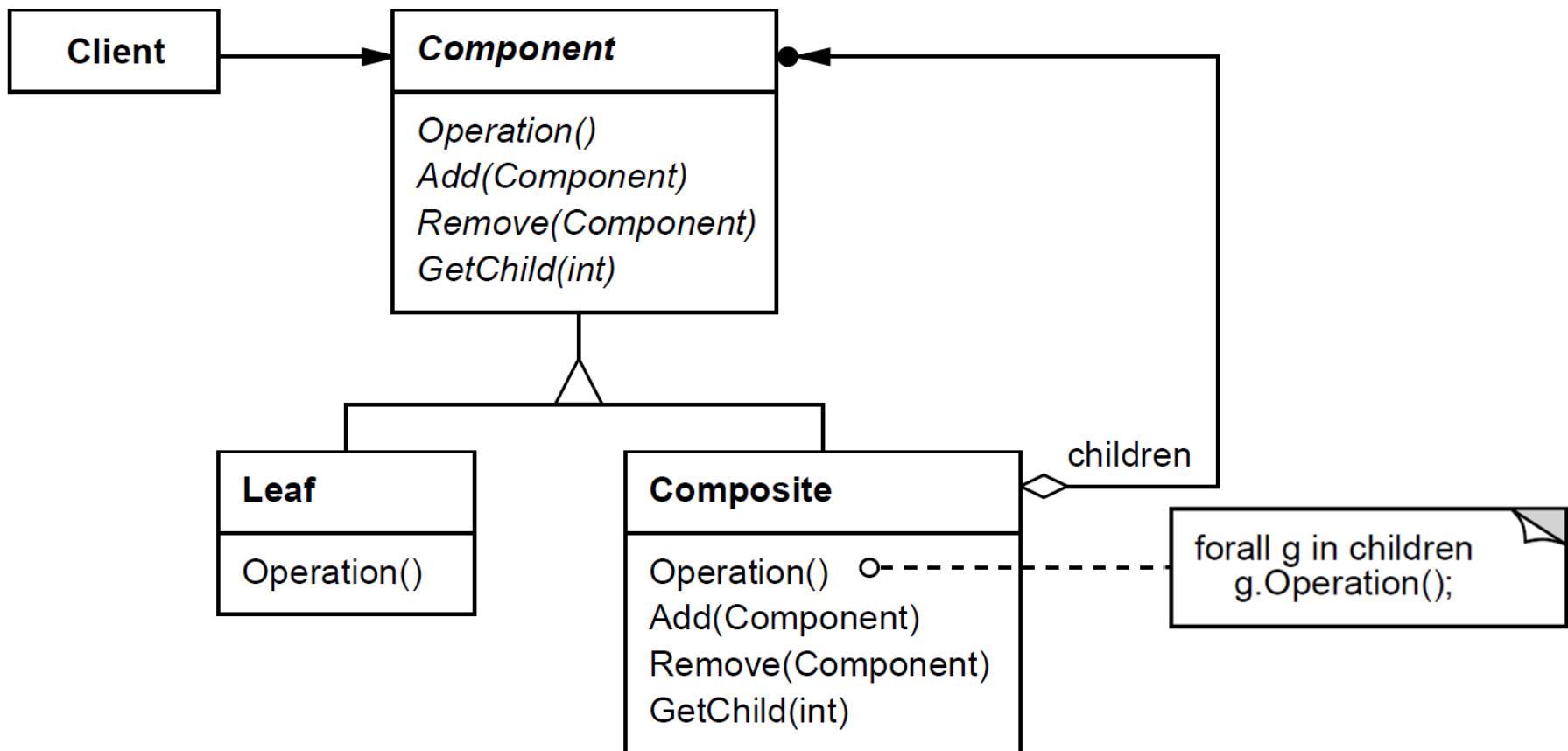
Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives

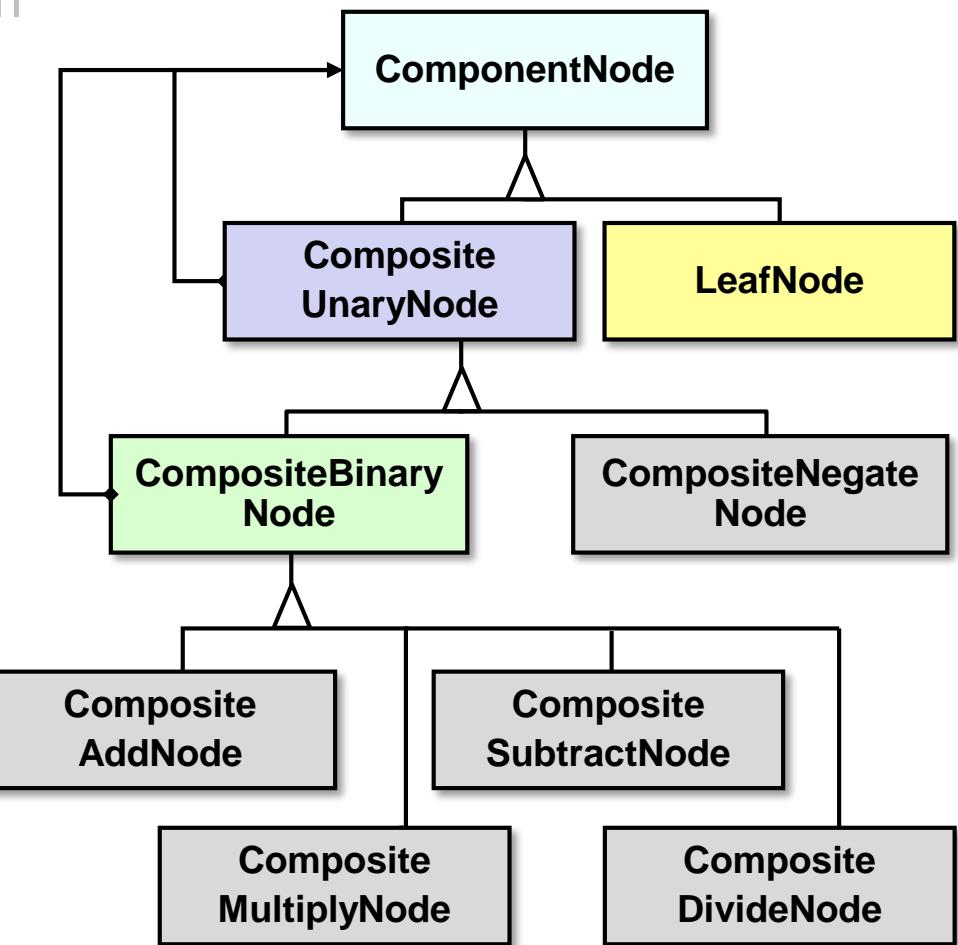
Learning Objectives

- Understand the *Composite* pattern



Learning Objectives

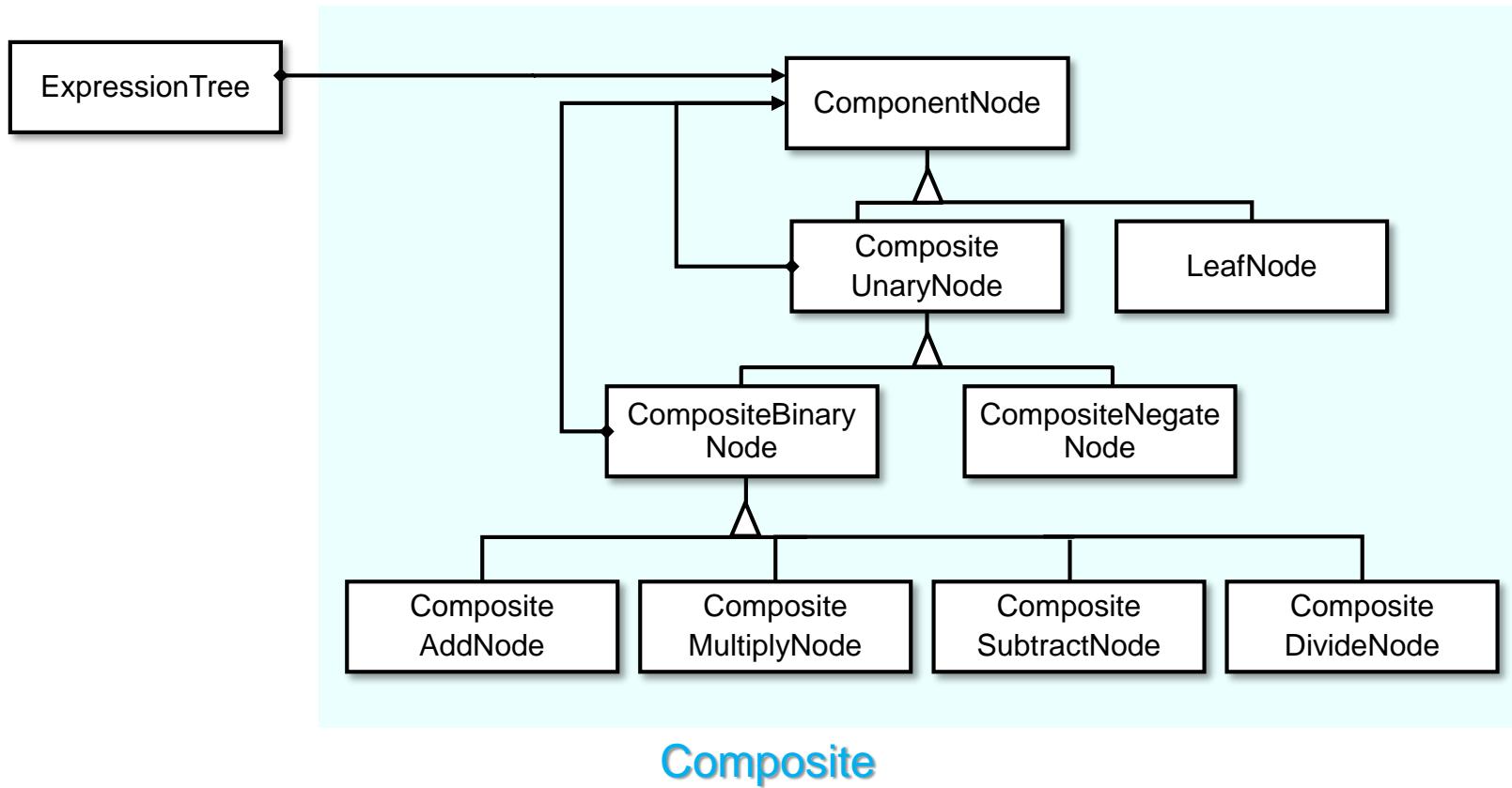
- Understand the *Composite* pattern
- Recognize how *Composite* can be applied to make the expression tree more uniform & extensible



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

A Pattern for Structuring the Expression Tree

Purpose: Define the key internal data structure for the expression tree

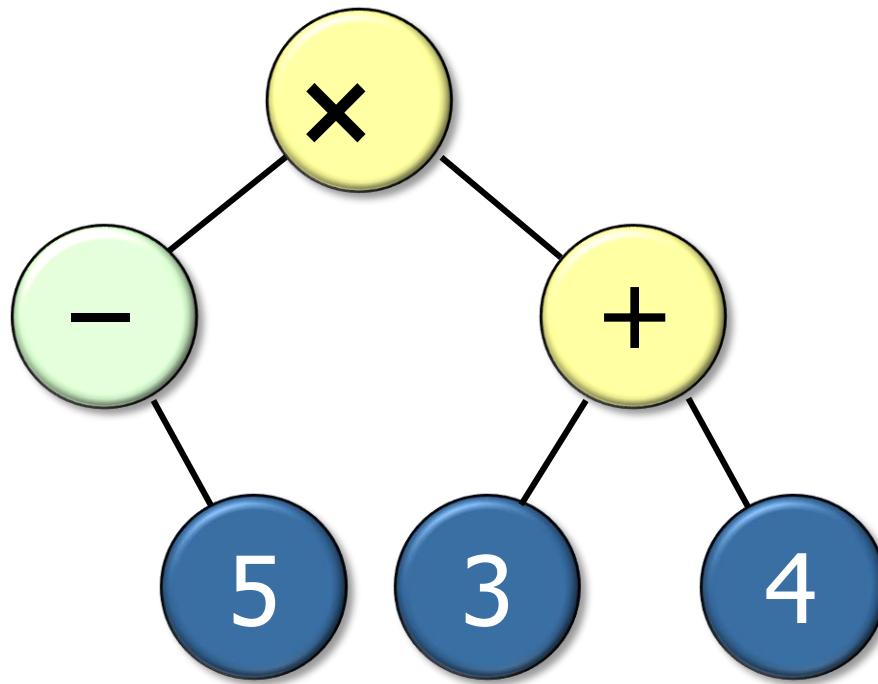


Composite

Composite simplifies adding new types of nodes (& new node operations)

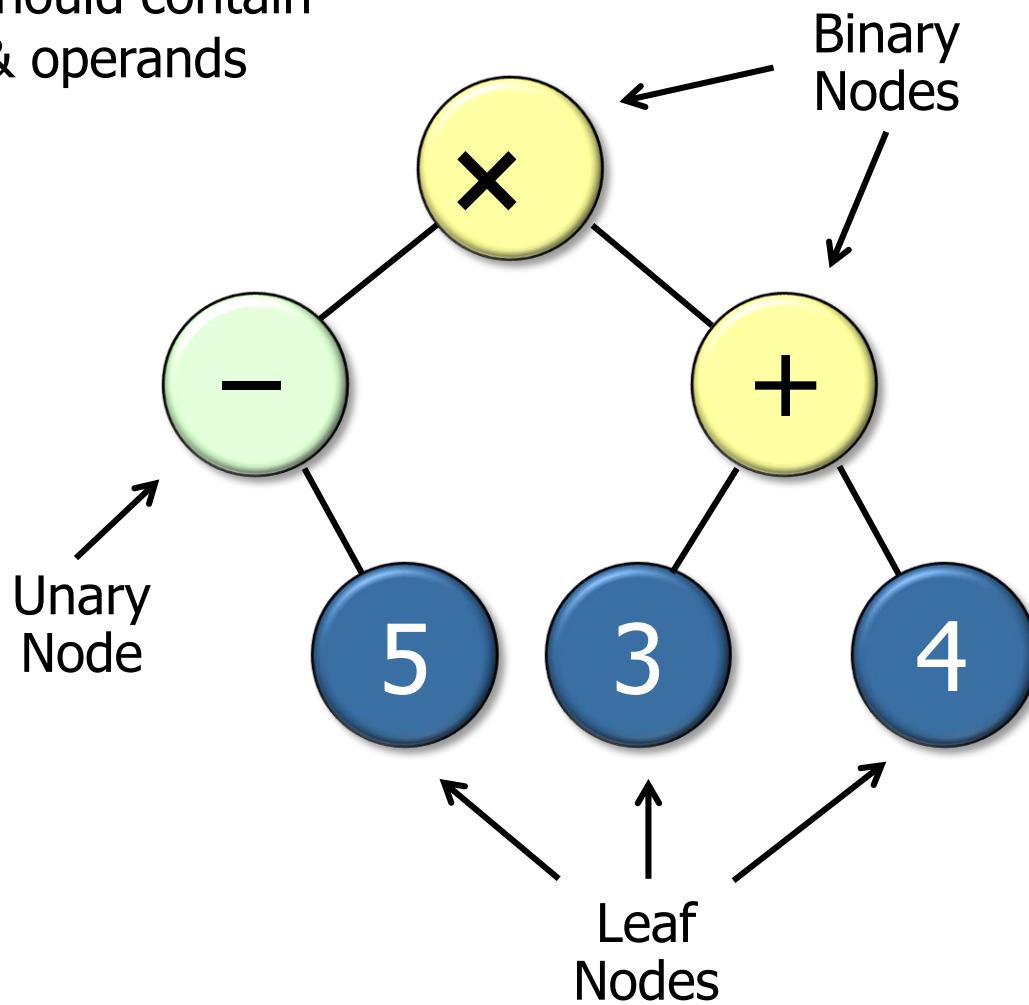
Context: OO Expression Tree Processing App

- The design of an expression tree should reflect its “physical” structure



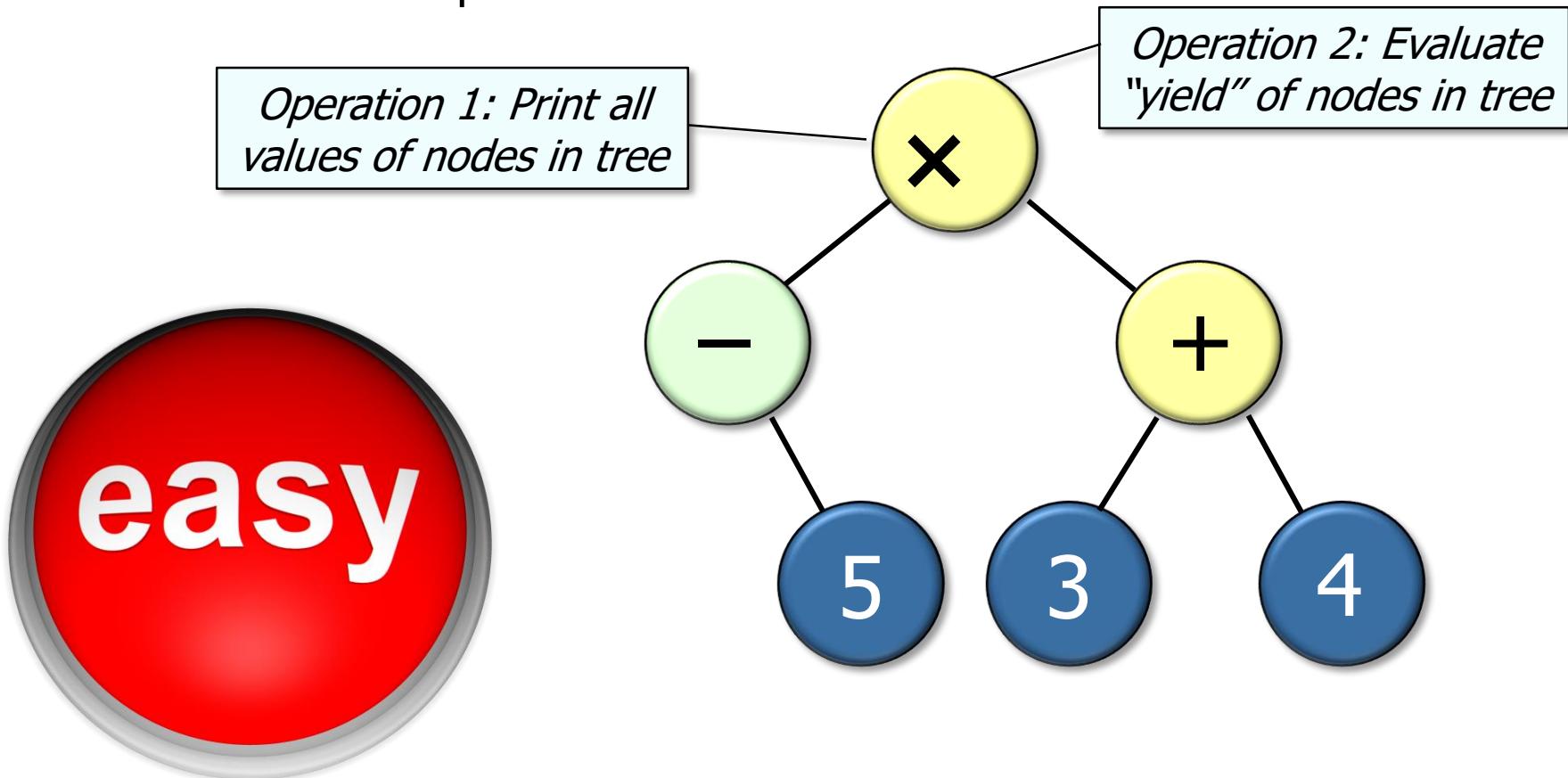
Context: OO Expression Tree Processing App

- The design of an expression tree should reflect its “physical” structure
 - e.g., the tree structure should contain binary/unary operators & operands



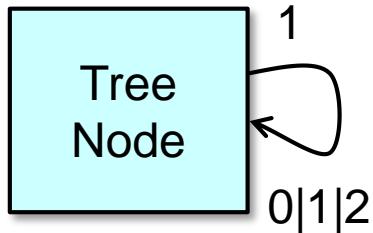
Context: OO Expression Tree Processing App

- Adding new operations on tree nodes should require little/no modifications to the tree's structure & implementation



Problem: Non-extensible & Error-prone Designs

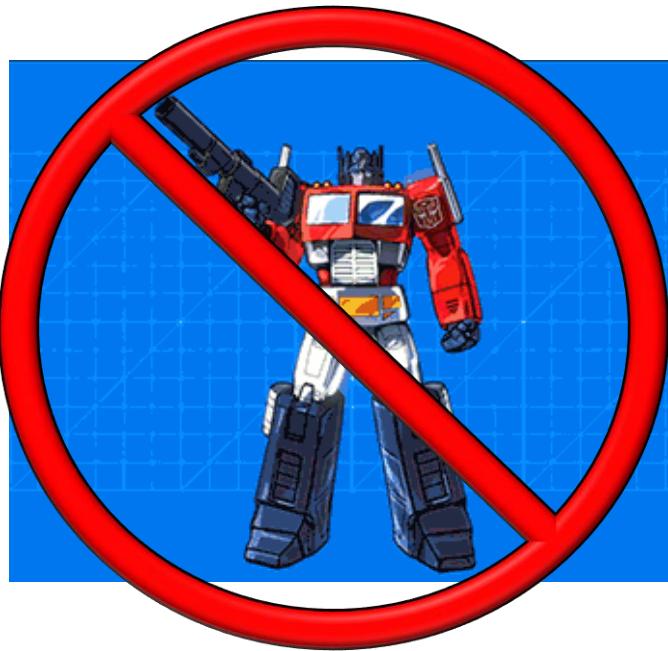
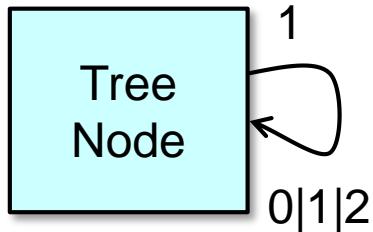
- Tightly coupling expression tree data structures & functionality impedes extensibility



```
typedef struct TreeNode {  
    enum { NUM, UNARY, BINARY } tag_;  
    short use_;  
    union {  
        char op_[3]; int num_;  
    } o_;  
#define num_ o_.num_  
#define op_ o_.op_  
    union {  
        struct TreeNode *unary_;  
        struct { struct TreeNode *l_,  
                 *r_;} binary_;  
    } c_;  
#define unary_ c_.unary_  
#define binary_ c_.binary_  
} TreeNode;
```

Problem: Non-extensible & Error-prone Designs

- Tightly coupling expression tree data structures & functionality impedes extensibility

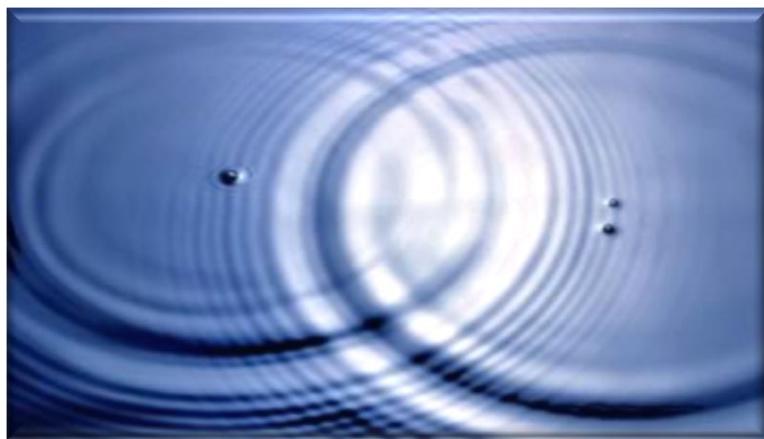


```
typedef struct TreeNode {  
    enum { NUM, UNARY, BINARY } tag_;  
    short use_;  
    union {  
        char op_[3]; int num_;  
    } o_;  
#define num_ o_.num_  
#define op_ o_.op_  
    union {  
        struct TreeNode *unary_;  
        struct { struct TreeNode *l_,  
                 *r_;} binary_;  
    } c_;  
#define unary_ c_.unary_  
#define binary_ c_.binary_  
} TreeNode;
```

Lack of extensibility is a major limitation with algorithmic decomposition

Problem: Non-extensible & Error-prone Designs

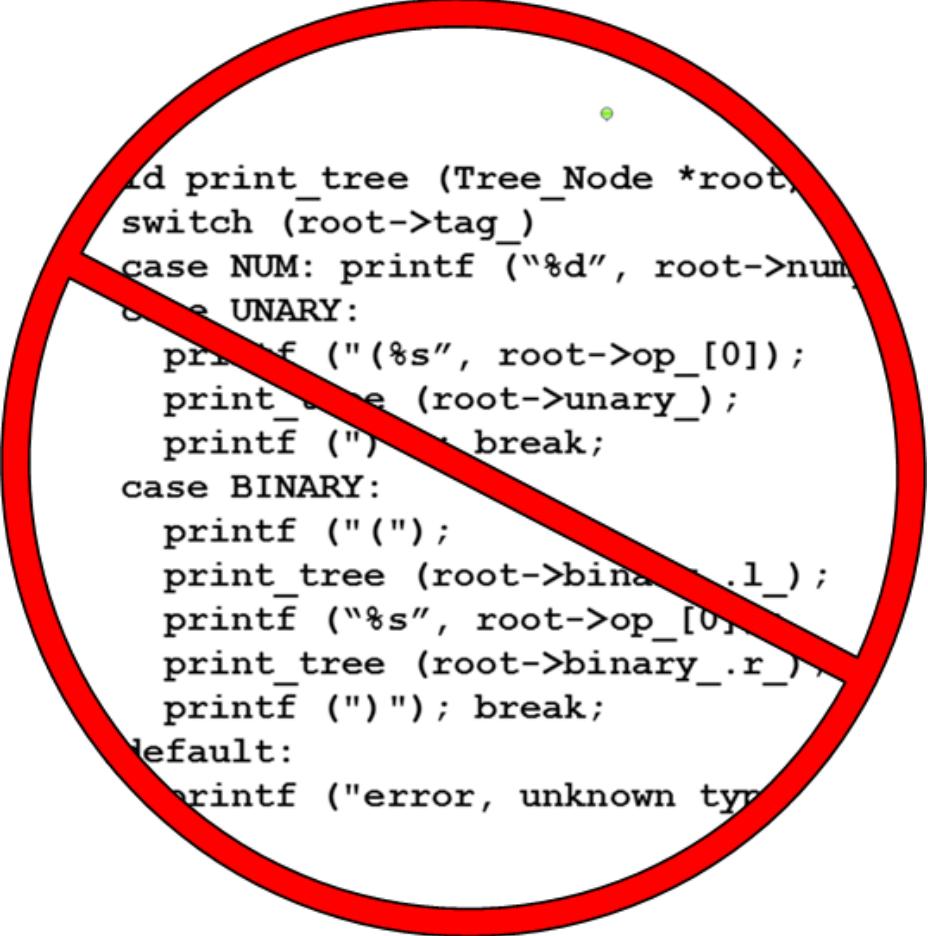
- Tightly coupling expression tree data structures & functionality impedes extensibility
 - e.g., adding new types of nodes or new node operations affects many parts of the program



```
typedef struct TreeNode {  
    enum { NUM, UNARY, BINARY,  
          TERNARY } tag_;  
  
    union {  
        char op_[4];  
        int num_;  
    } o_;  
    ...  
    union {  
        ...  
        struct {  
            Tree_Node *l_,  
            *m_,  
            *r_;  
        } ternary_;  
    } c_;  
#define ternary_ c_.ternary_  
} TreeNode;
```

Problem: Non-extensible & Error-prone Designs

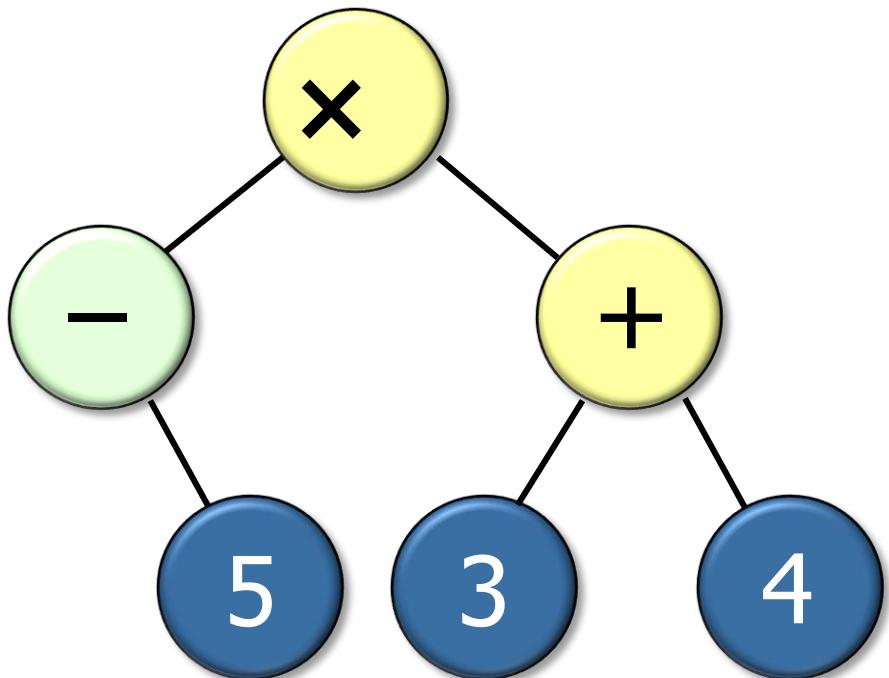
- Differentiating operators & operands via type tags & switch statements is tedious & error-prone to program & maintain



```
ld print_tree (Tree_Node *root,
switch (root->tag_)
case NUM: printf ("%d", root->num);
case UNARY:
    printf ("%s", root->op_[0]);
    print_tree (root->unary_);
    printf ("");
    break;
case BINARY:
    printf "(");
    print_tree (root->binary_.l_);
    printf ("%s", root->op_[0]);
    print_tree (root->binary_.r_);
    printf ")"); break;
default:
    printf ("error, unknown type");
}
```

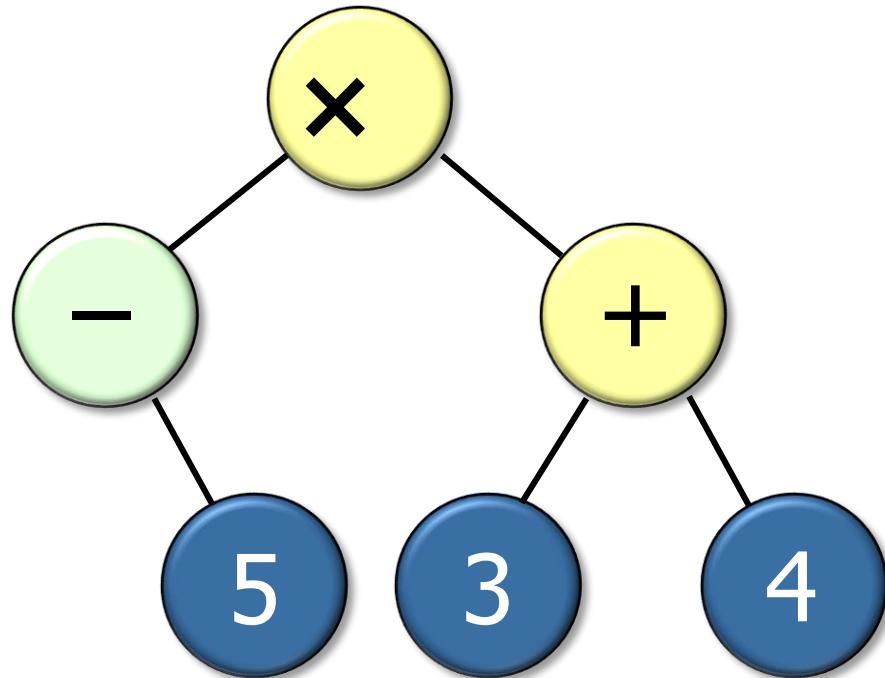
Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes



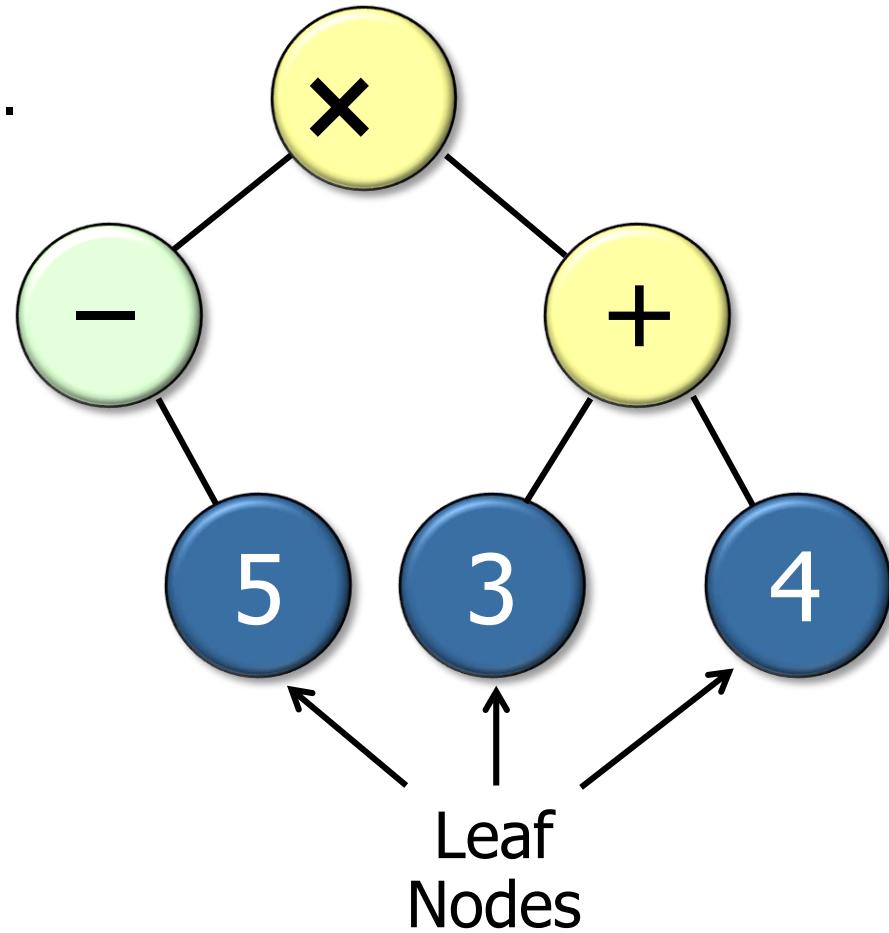
Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes, e.g.
 - Structure nodes into a hierarchy that captures properties of each node



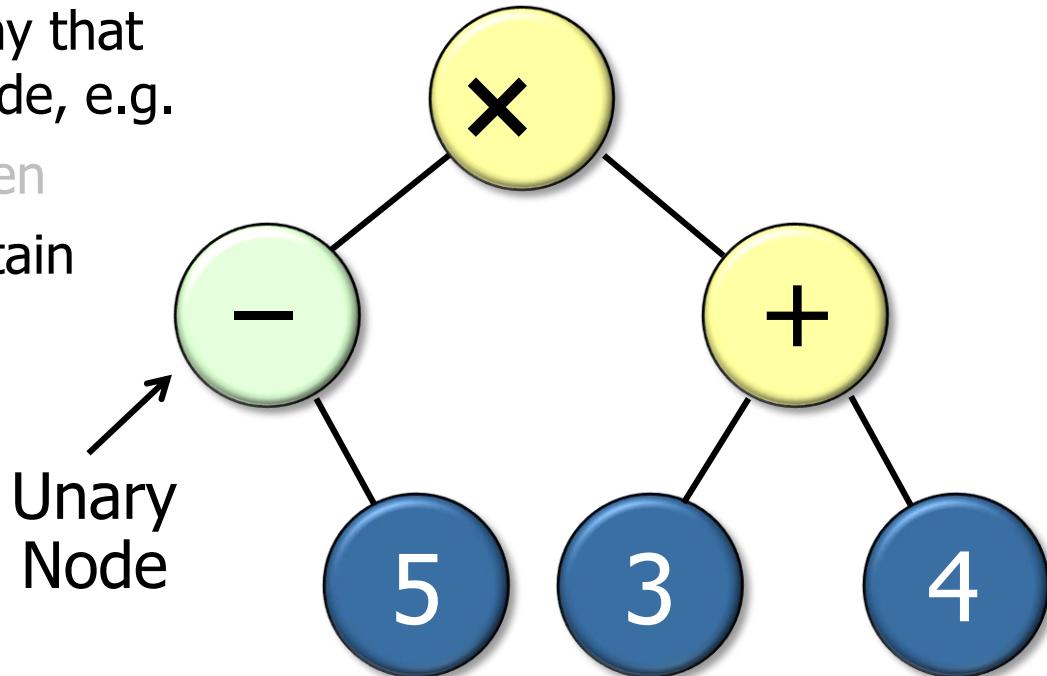
Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes, e.g.
 - Structure nodes into a hierarchy that captures properties of each node, e.g.
 - Leaf nodes contain no children



Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes, e.g.
 - Structure nodes into a hierarchy that captures properties of each node, e.g.
 - Leaf nodes contain no children
 - Unary nodes recursively contain one child node

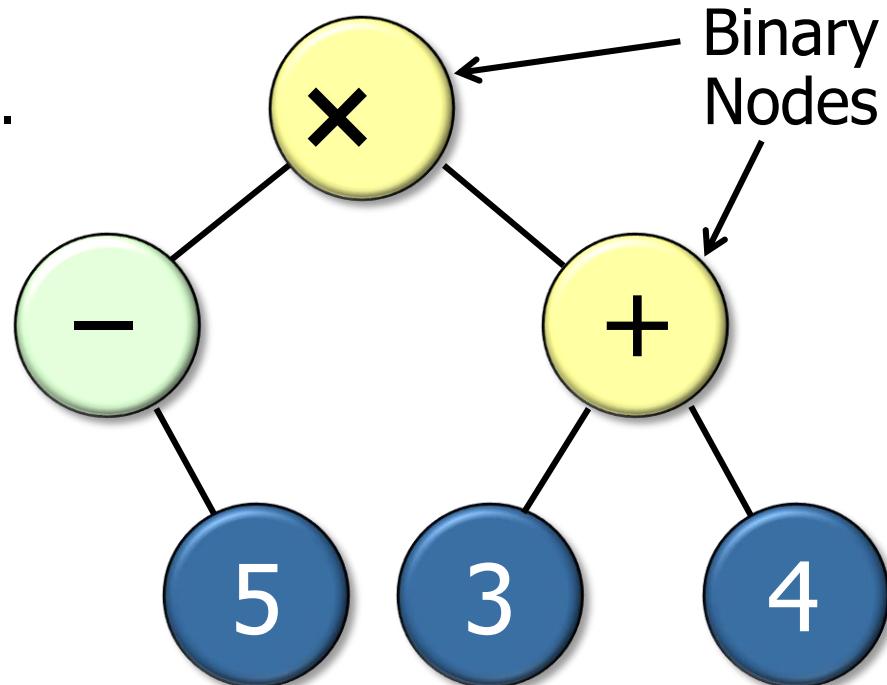


Solution: Recursive Object Structure

- Model an expression tree as a recursive collection of nodes, e.g.

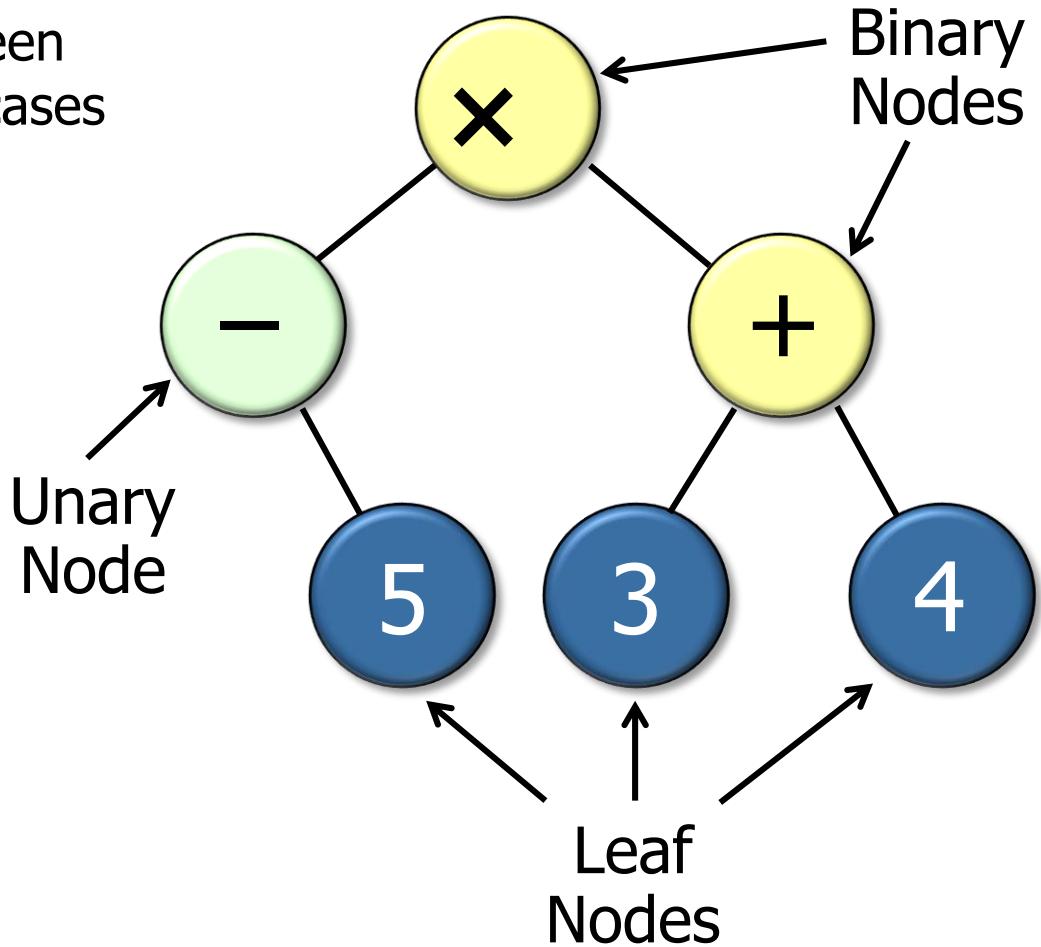
- Structure nodes into a hierarchy that captures properties of each node, e.g.

- Leaf nodes contain no children
 - Unary nodes recursively contain one child node
 - Binary nodes recursively contain two children nodes



Solution: Recursive Object Structure

- Treat operators & operands uniformly
 - e.g., minimize distinction between one vs. many to avoid special cases



ComponentNode Class Overview

- Abstract super class for composable expression tree node objects

Class methods

int	<u>getItem()</u>
<u>ComponentNode</u>	<u>getLeftChild()</u>
<u>ComponentNode</u>	<u>getRightChild()</u>
void	<u>accept(Visitor visitor)</u>

ComponentNode Class Overview

- Abstract super class for composable expression tree node objects

Class methods

These methods access relevant fields
(may be no-ops for some subclasses)

int	<code>getItem()</code>	
<u>ComponentNode</u>	<code>getLeftChild()</code>	
<u>ComponentNode</u>	<code>getRightChild()</code>	
void	<code>accept(Visitor visitor)</code>	

ComponentNode Class Overview

- Abstract super class for composable expression tree node objects

Class methods

<code>int <u>getItem()</u></code> <code><u>ComponentNode</u></code>	<code><u>getLeftChild()</u></code>
<code><u>ComponentNode</u></code>	<code><u>getRightChild()</u></code>
<code>void <u>accept(Visitor visitor)</u></code>	



This hook method plays an essential role in *Iterator* & *Visitor* patterns (covered later)

ComponentNode Class Overview

- Abstract super class for composable expression tree node objects

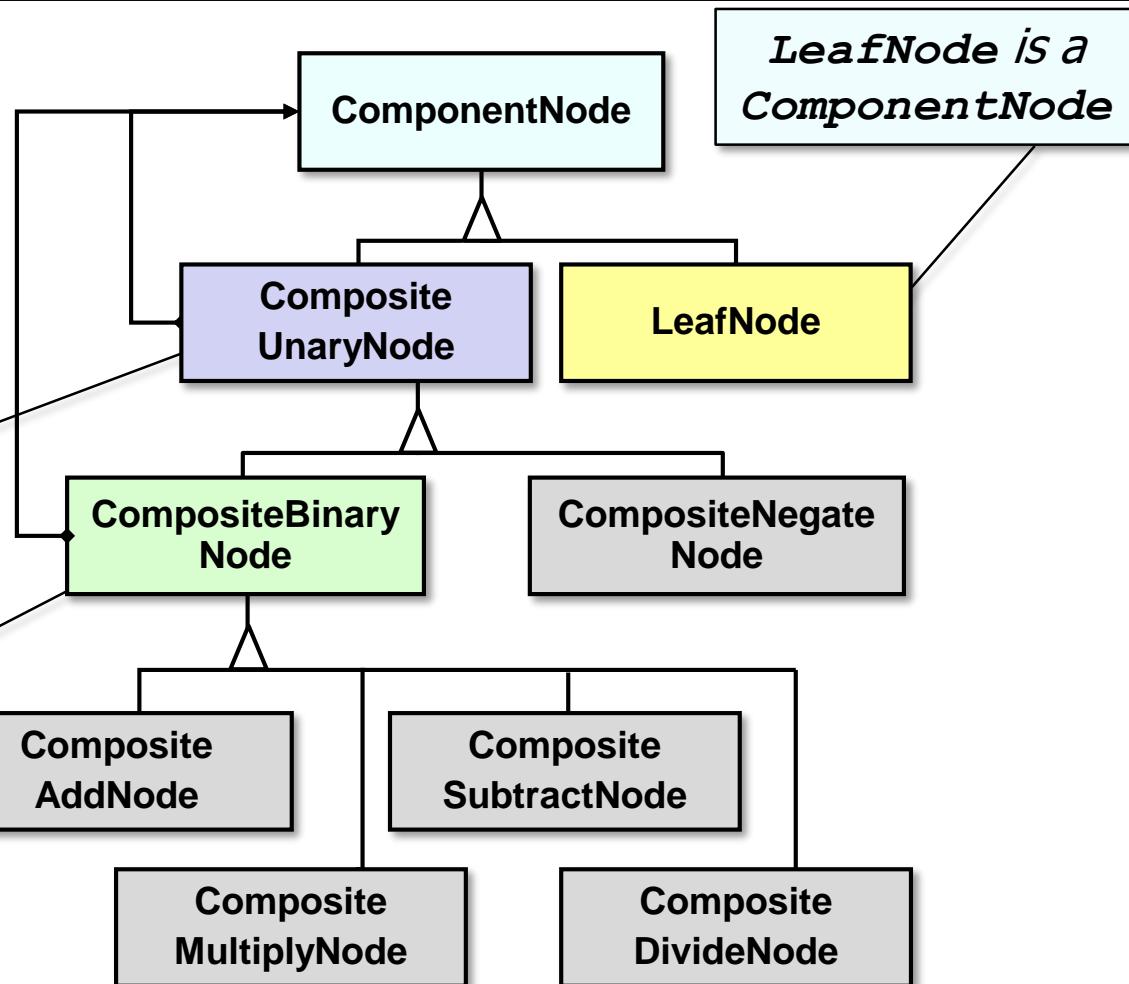
Class methods

int	<u>getItem()</u>
<u>ComponentNode</u>	<u>getLeftChild()</u>
<u>ComponentNode</u>	<u>getRightChild()</u>
void	<u>accept(Visitor visitor)</u>

- Commonality:** Super class interface used by all nodes in an expression tree
- Variability:** Each subclass defines state & method implementations that can be customized for specific types of expression tree nodes

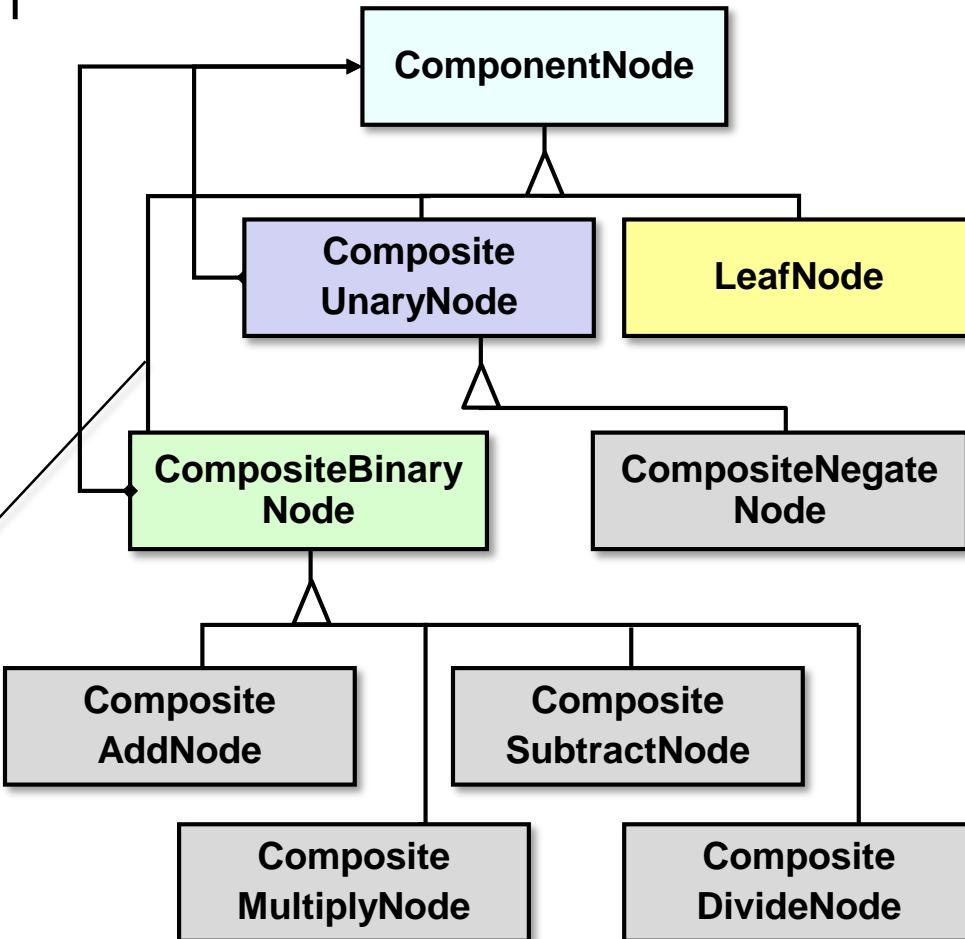
ComponentNode Class Hierarchy Overview

- Note the inherent recursion in this hierarchy



ComponentNode Class Hierarchy Overview

- Note the inherent recursion in this hierarchy



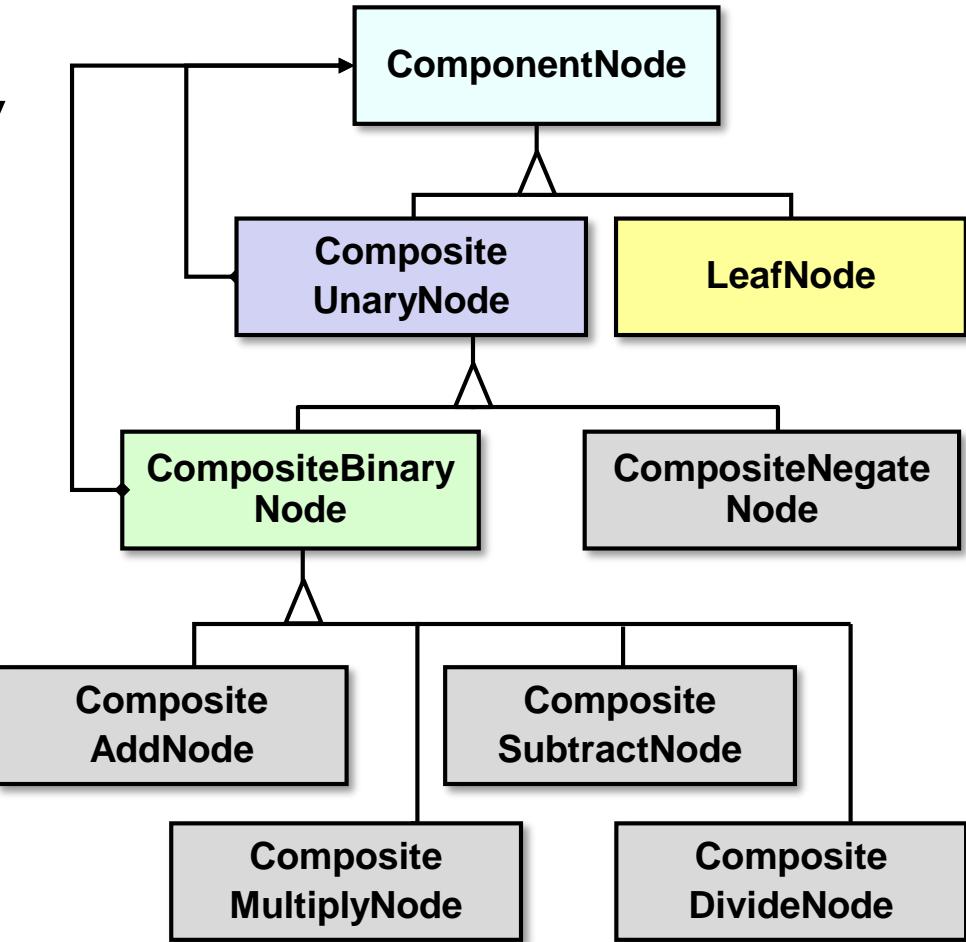
Elements of the Composite Pattern

Composite

GoF Object structural

Intent

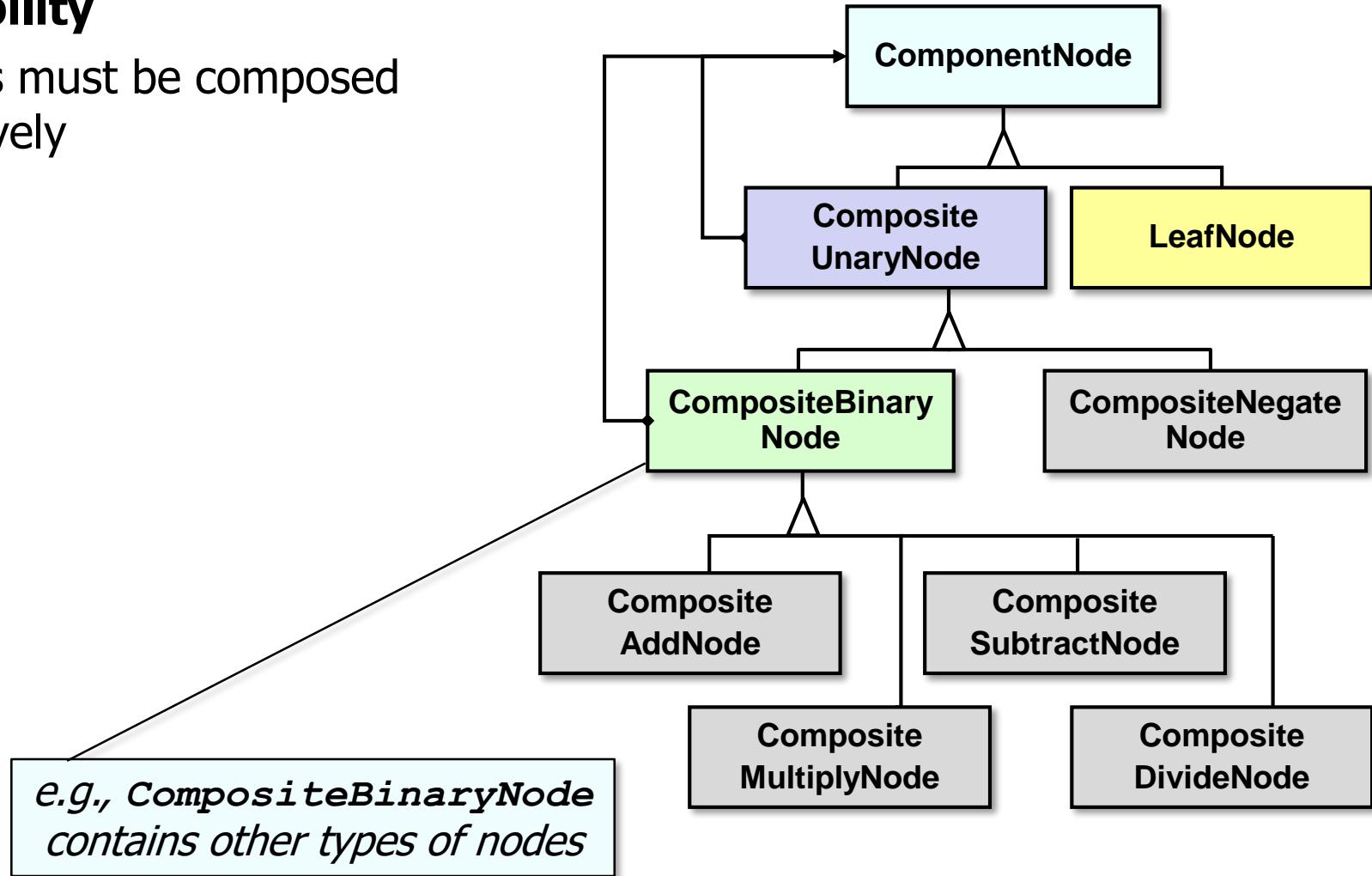
- Treat individual objects & multiple, recursively-composed objects uniformly



See en.wikipedia.org/wiki/Composite_pattern

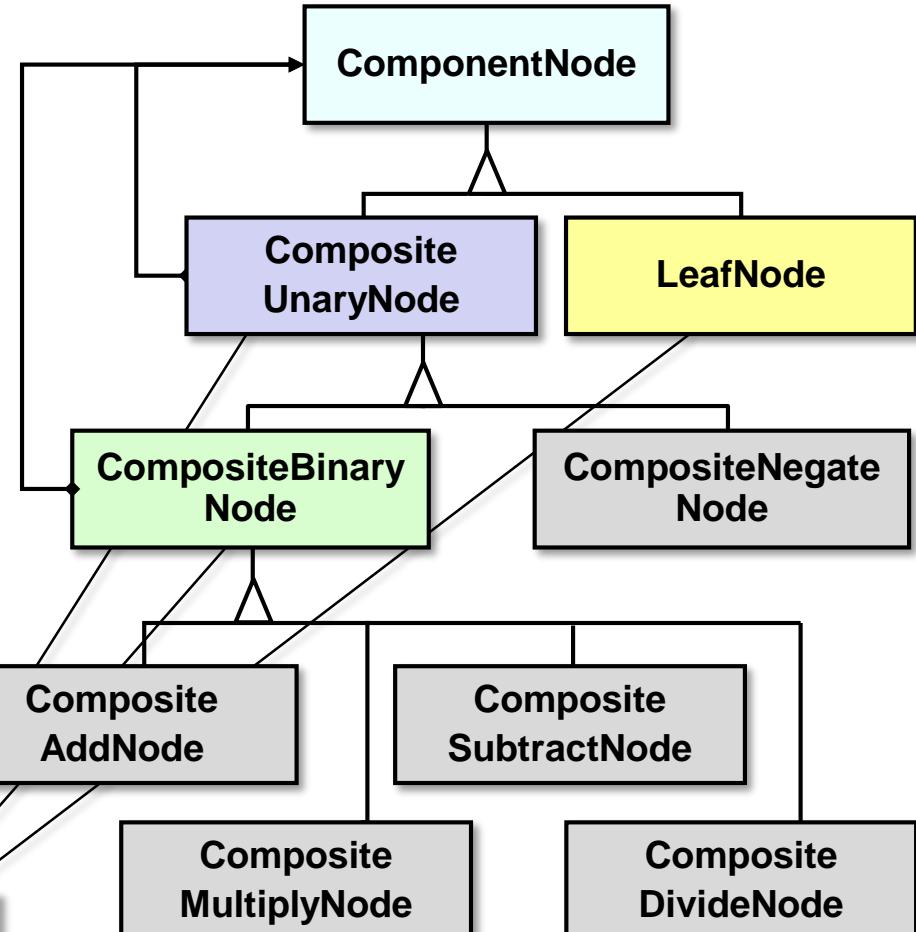
Applicability

- Objects must be composed recursively



Applicability

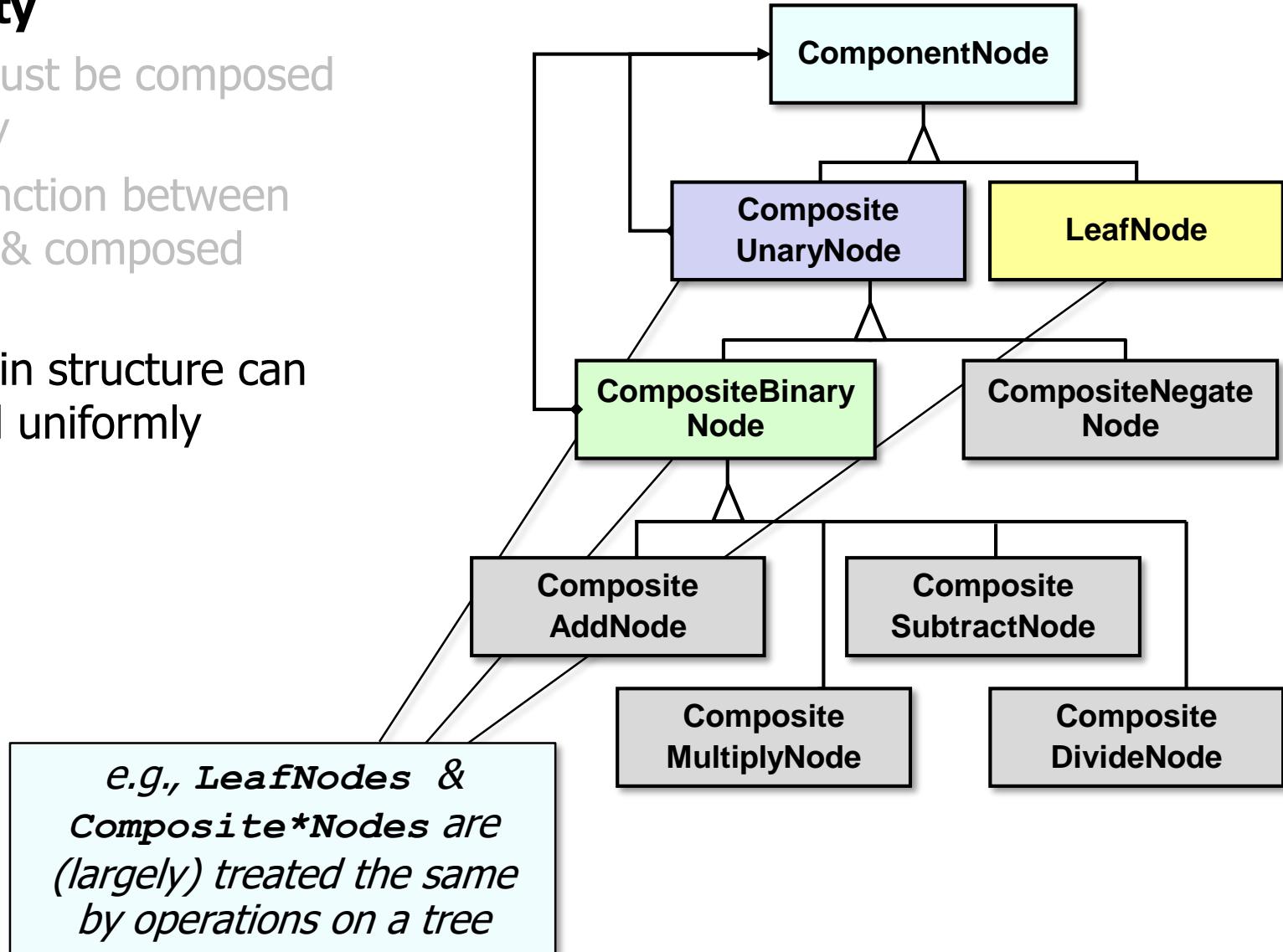
- Objects must be composed recursively
- & no distinction between individual & composed elements



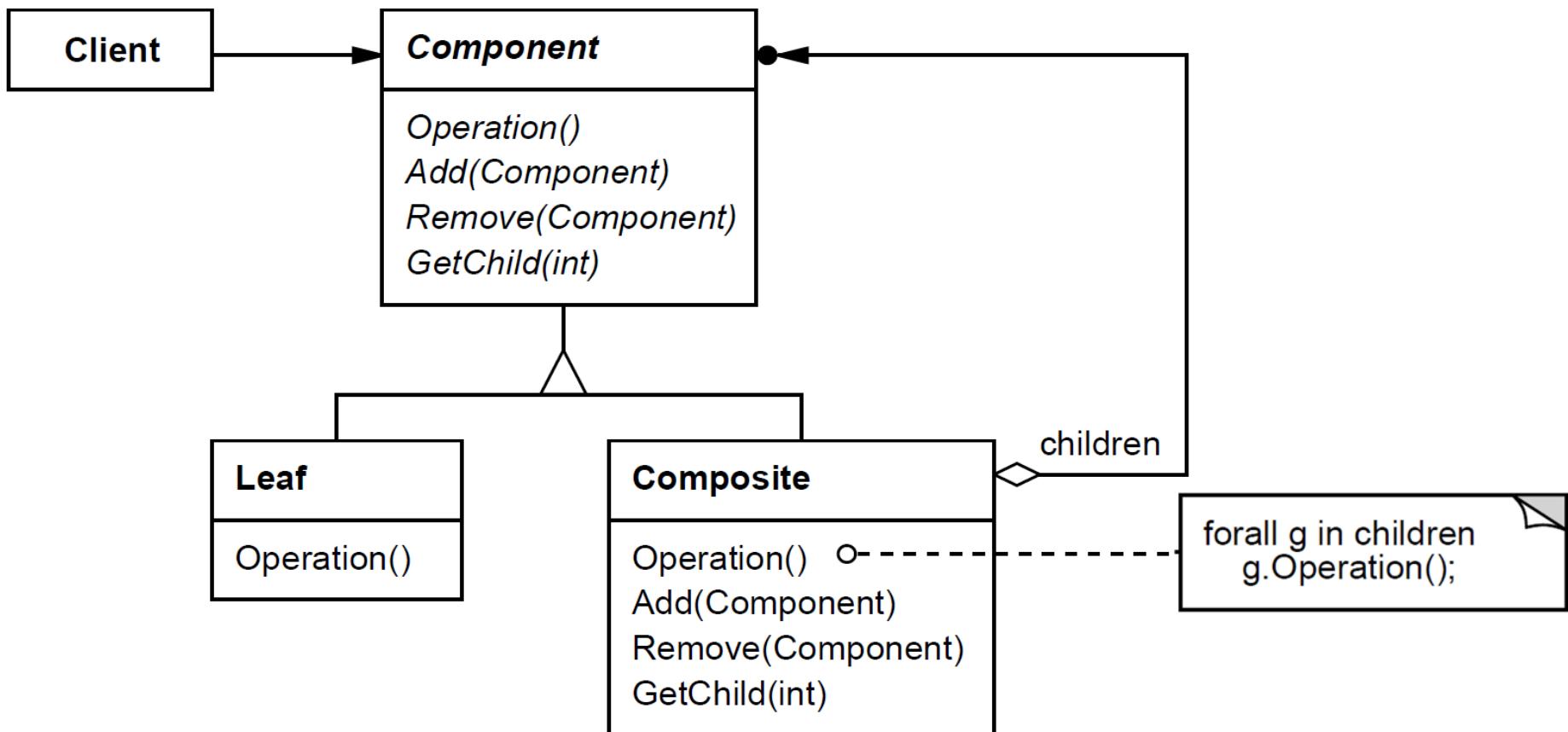
*e.g., LeafNodes &
Composite*Nodes
have the same API*

Applicability

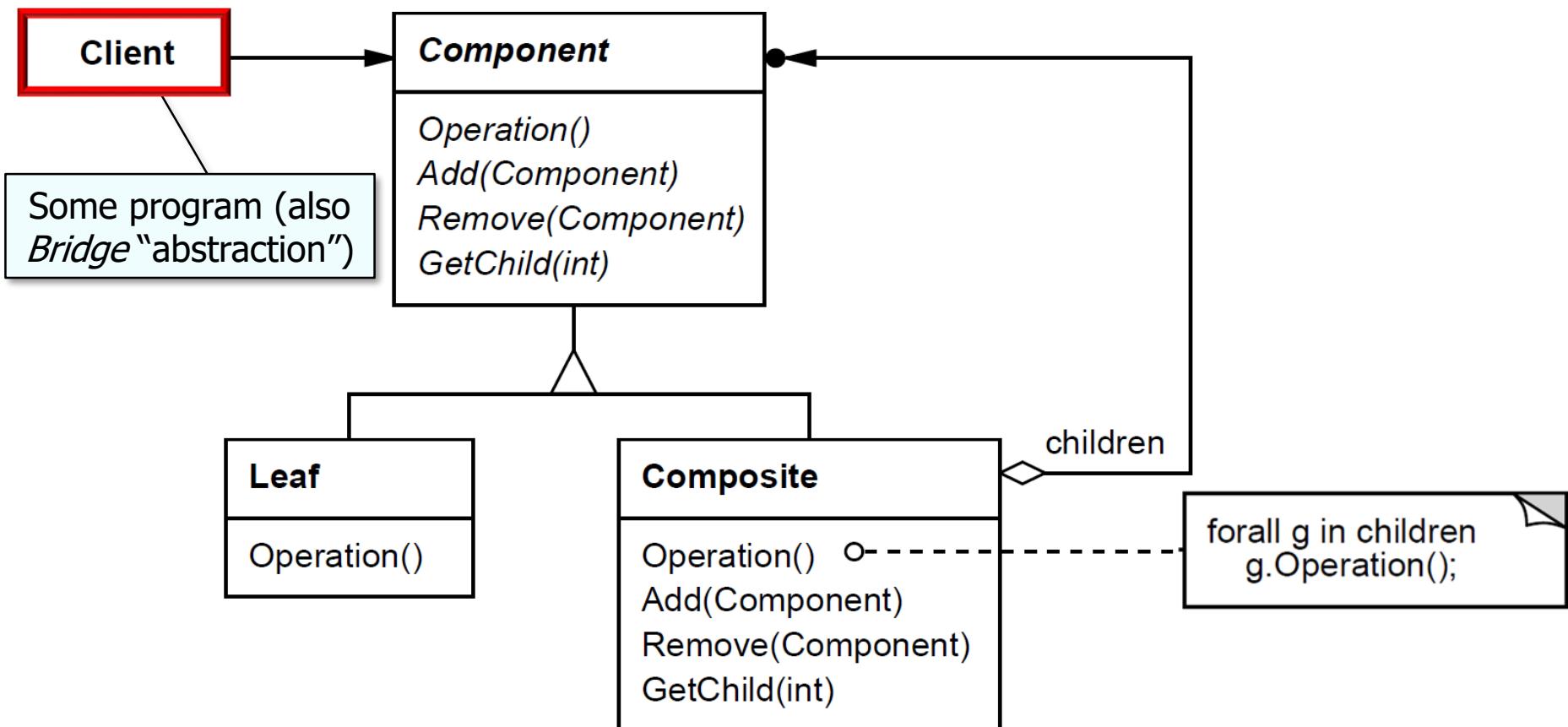
- Objects must be composed recursively
- & no distinction between individual & composed elements
- & objects in structure can be treated uniformly



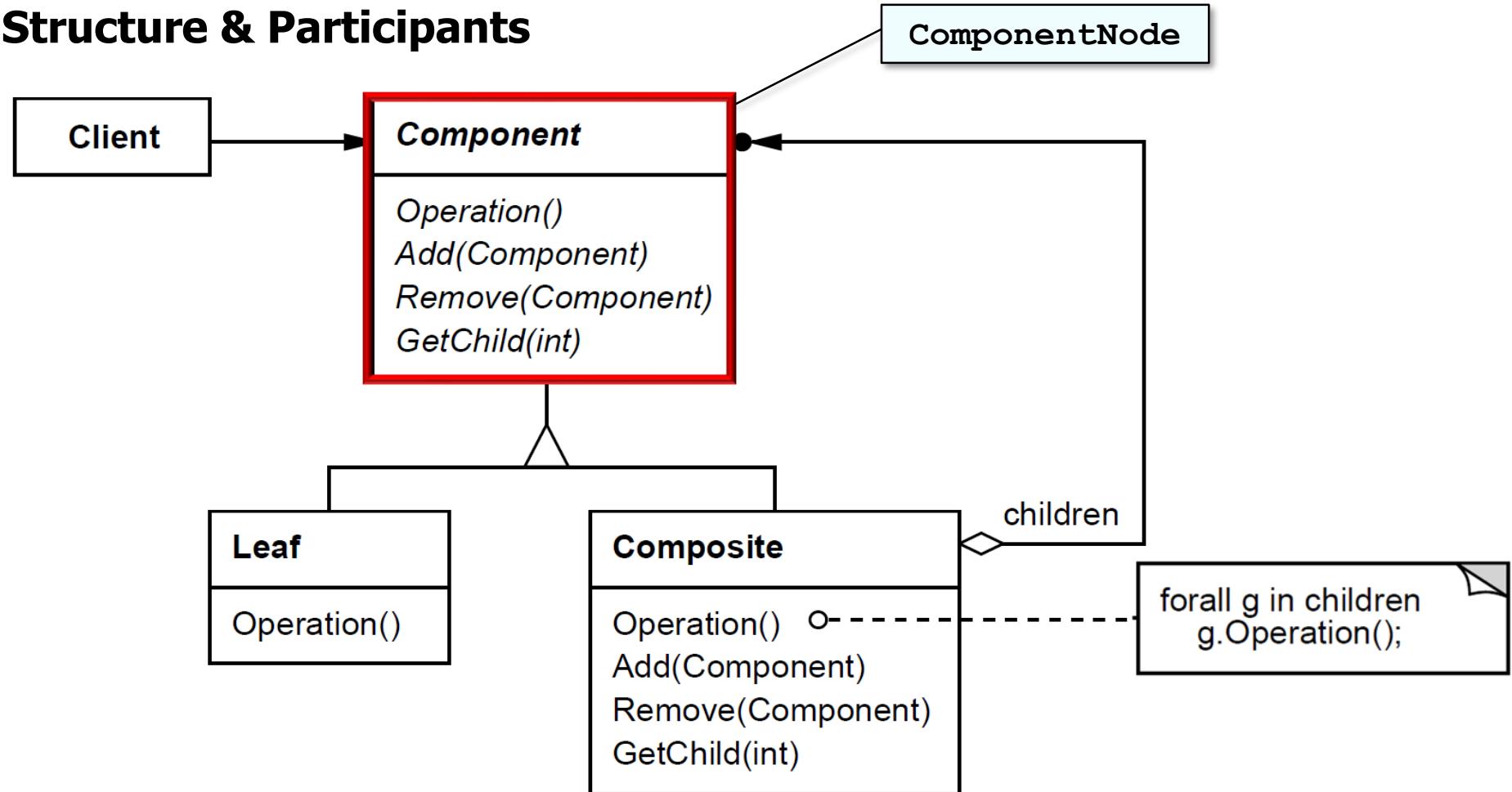
Structure & Participants



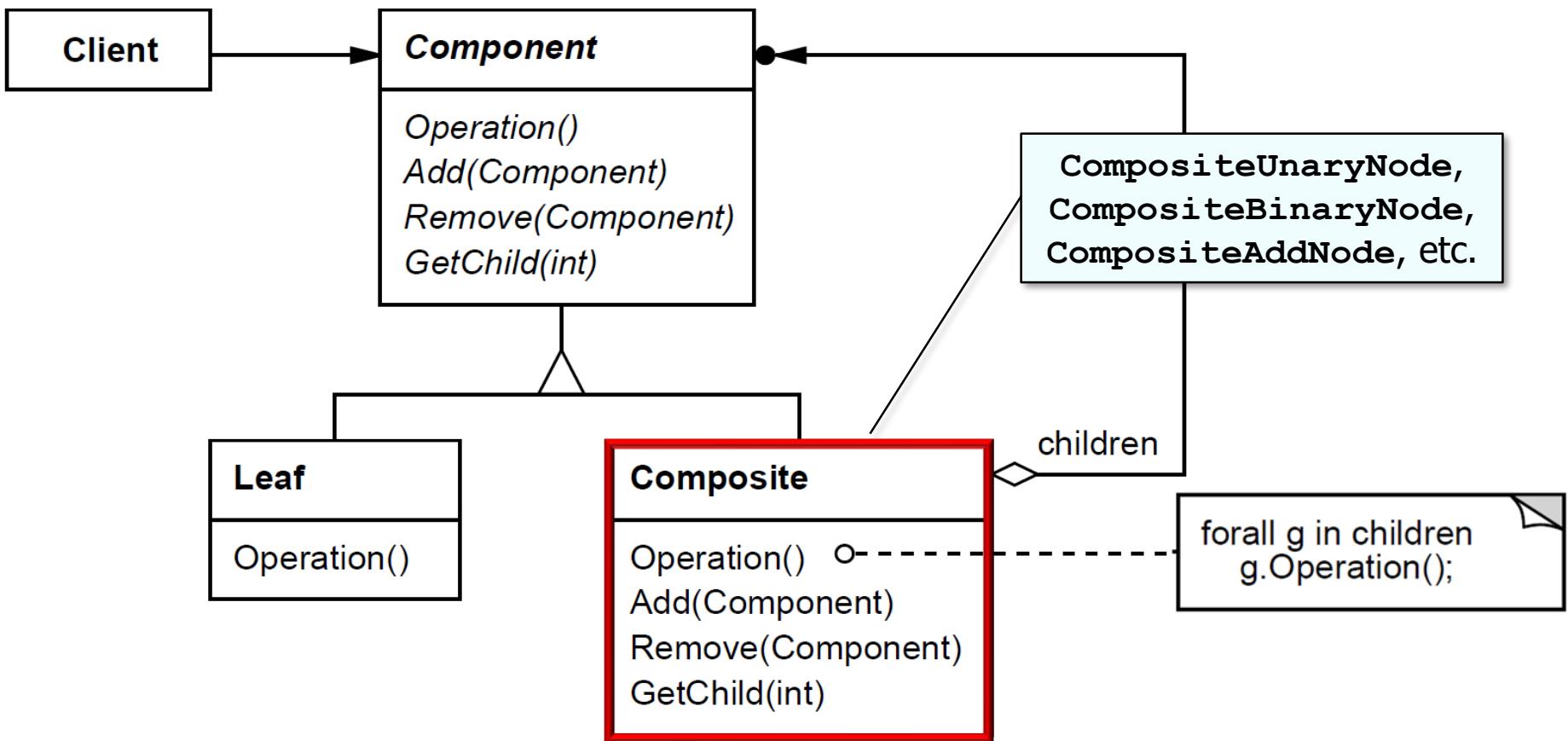
Structure & Participants



Structure & Participants



Structure & Participants



Structure & Participants

Client

Component

*Operation()
Add(Component)
Remove(Component)
GetChild(int)*

Leaf

Operation()

LeafNode

Composite

*Operation() O-
Add(Component)
Remove(Component)
GetChild(int)*

children

forall g in children
g.Operation();

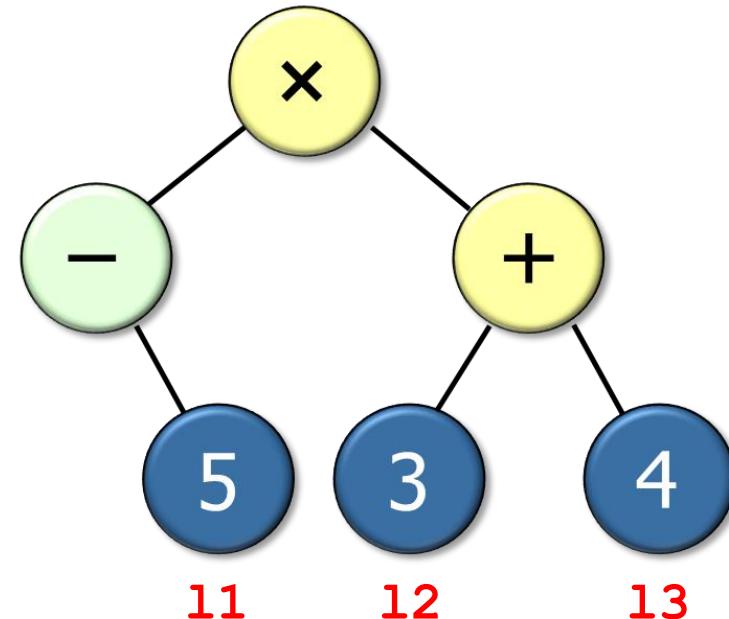
Composite example in Java

- Build an expression tree based on recursively-composed objects

```
ComponentNode 11 =  
    new LeafNode(5);
```

```
ComponentNode 12 =  
    new LeafNode(3);
```

```
ComponentNode 13 =  
    new LeafNode(4);
```



Composite example in Java

- Build an expression tree based on recursively-composed objects

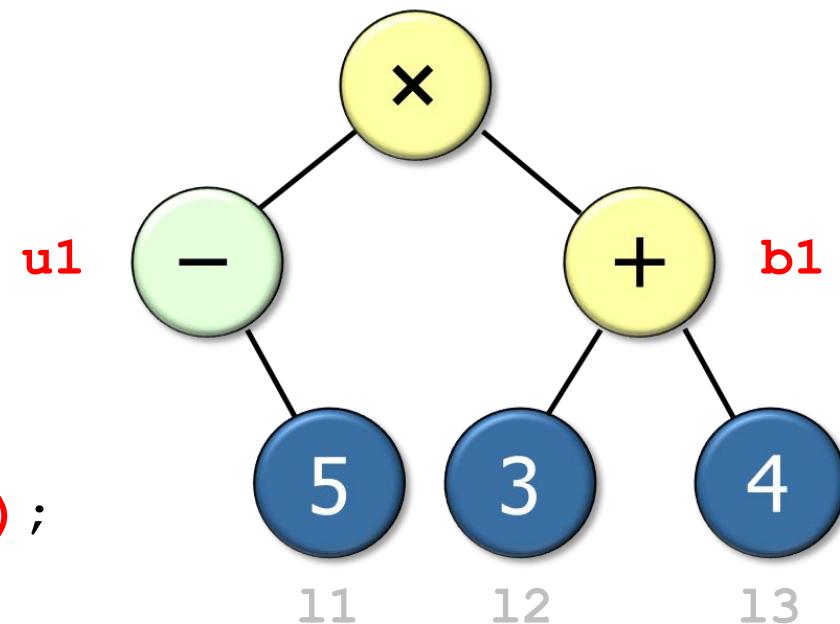
```
ComponentNode 11 =  
    new LeafNode(5);
```

```
ComponentNode 12 =  
    new LeafNode(3);
```

```
ComponentNode 13 =  
    new LeafNode(4);
```

```
ComponentNode u1 =  
    new CompositeNegateNode(11);
```

```
ComponentNode b1 =  
    new CompositeAddNode(12, 13);
```



Composite example in Java

- Build an expression tree based on recursively-composed objects

```
ComponentNode 11 =  
    new LeafNode(5);
```

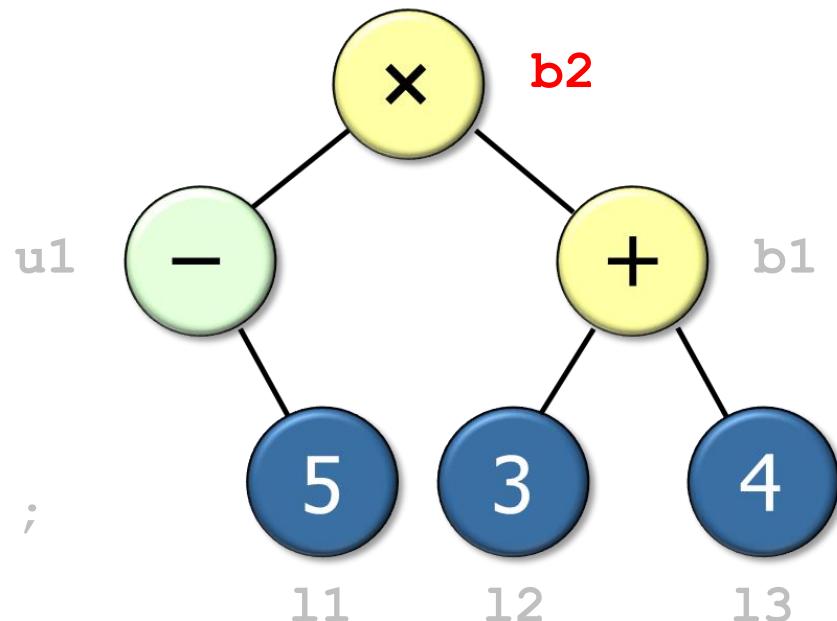
```
ComponentNode 12 =  
    new LeafNode(3);
```

```
ComponentNode 13 =  
    new LeafNode(4);
```

```
ComponentNode u1 =  
    new CompositeNegateNode(11);
```

```
ComponentNode b1 =  
    new CompositeAddNode(12, 13);
```

```
ComponentNode b2 =  
    new CompositeMultiplyNode(u1, b1);
```



Composite example in Java

- Build an expression tree based on recursively-composed objects

```
ComponentNode 11 =  
    new LeafNode(5);
```

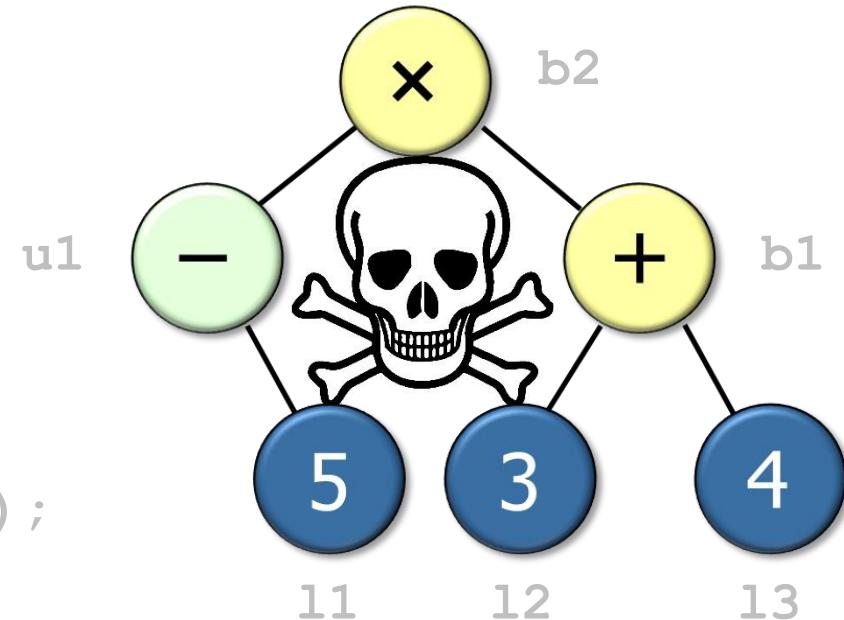
```
ComponentNode 12 =  
    new LeafNode(3);
```

```
ComponentNode 13 =  
    new LeafNode(4);
```

```
ComponentNode u1 =  
    new CompositeNegateNode(11);
```

```
ComponentNode b1 =  
    new CompositeAddNode(12, 13);
```

```
ComponentNode b2 =  
    new CompositeMultiplyNode(u1, b1);
```



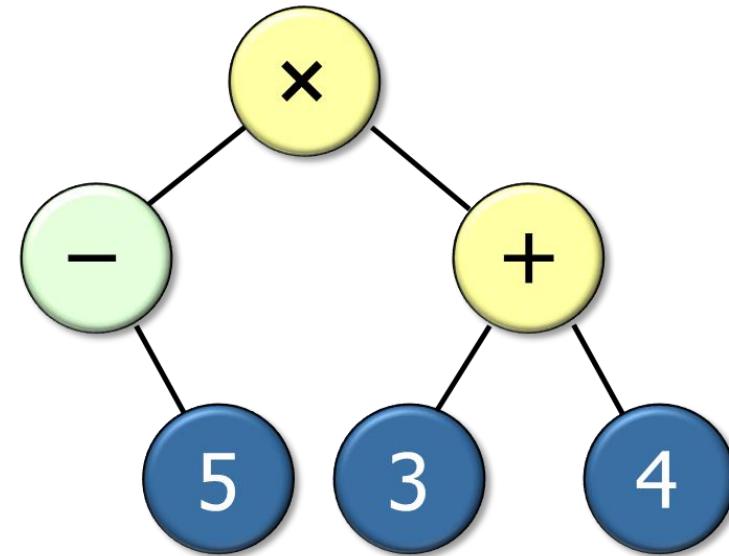
Java's garbage collector deletes dynamically allocated memory automatically when it's no longer needed

Composite example in Java

- Build an expression tree based on recursively-composed objects

```
ComponentNode exprTree =  
makeExpressionTree  
("-5 * (3 + 4)");
```

A better way to build an expression tree is via a Creational pattern



Consequences

+ Uniformity

- Treat components the same regardless of complexity & behavior

```
ExpressionTree exprTree = ...;  
Visitor visitor = ...;  
  
for (Iterator<ExpressionTree> iter =  
      exprTree.iterator  
      (traversalOrder);  
      iter.hasNext());  
  
    iter.next().accept(visitor);
```

No syntactic distinction between leaf nodes or composite nodes (iterator variant)

Composite

GoF Object structural

Consequences

+ Uniformity

- Treat components the same regardless of complexity & behavior

```
ExpressionTree exprTree = ...;  
Visitor visitor = ...;  
  
for (Iterator<ExpressionTree> iter =  
      exprTree.iterator  
      (traversalOrder);  
      iter.hasNext());  
    iter.next().accept(visitor);
```

StreamUtils

```
.iteratorAsStream(iter, false)  
.forEach(exprTree ->  
  exprTree.accept(visitor));
```

Converts an iterator to a Java 8 stream

Consequences

+ Uniformity

- Treat components the same regardless of complexity & behavior

```
ExpressionTree exprTree = ...;  
Visitor visitor = ...;  
  
for (Iterator<ExpressionTree> iter =  
      exprTree.iterator  
      (traversalOrder);  
      iter.hasNext());  
    iter.next().accept(visitor);
```

StreamUtils

```
.iteratorAsStream(iter, false)  
.forEach(exprTree ->  
        exprTree.accept(visitor));
```

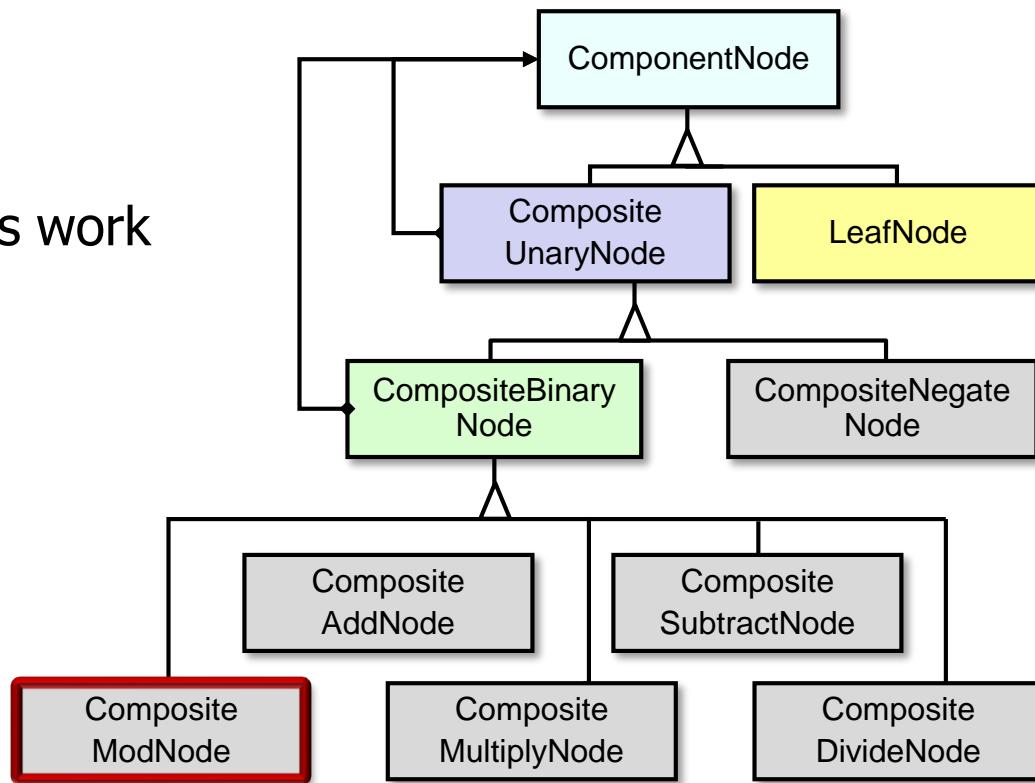
No syntactic distinction between leaf nodes or composite nodes (Java 8 streams variant)

Consequences

+ *Uniformity*

+ *Extensibility*

- New component subclasses work wherever existing ones do



Consequences

+ *Uniformity*

+ *Extensibility*

+ *Parsimony*

- Classes only include fields they need

```
public interface ComponentNode {  
    // No non-static fields & only  
    // default "no-op" methods  
  
    default int getItem() {  
        throw new  
            UnsupportedOperationException  
                ("method not implemented");  
    }  
  
    default ComponentNode getRightChild() {  
        return null;  
    }  
  
    default ComponentNode getLeftChild() {  
        return null;  
    }  
    ...  
}
```

See [ExpressionTree/CommandLine/src/expressiontree/nodes](#)

Consequences

+ Uniformity

+ Extensibility

+ Parsimony

- Classes only include fields & methods that they need

```
public class LeafNode
    extends ComponentNode {
    ...
    private int mItem;
    int getItem()
    { return mItem; }
}

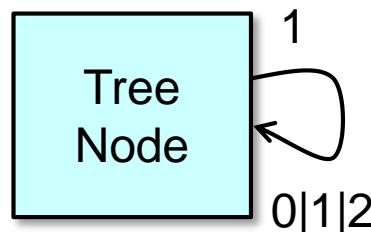
public class CompositeUnaryNode
    extends ComponentNode {
    ...
    // Reference to the right child.
    ComponentNode mRight;
    ComponentNode getRightChild()
    { return mRight; }
}
```

See [ExpressionTree/CommandLine/src/expressiontree/nodes](#)

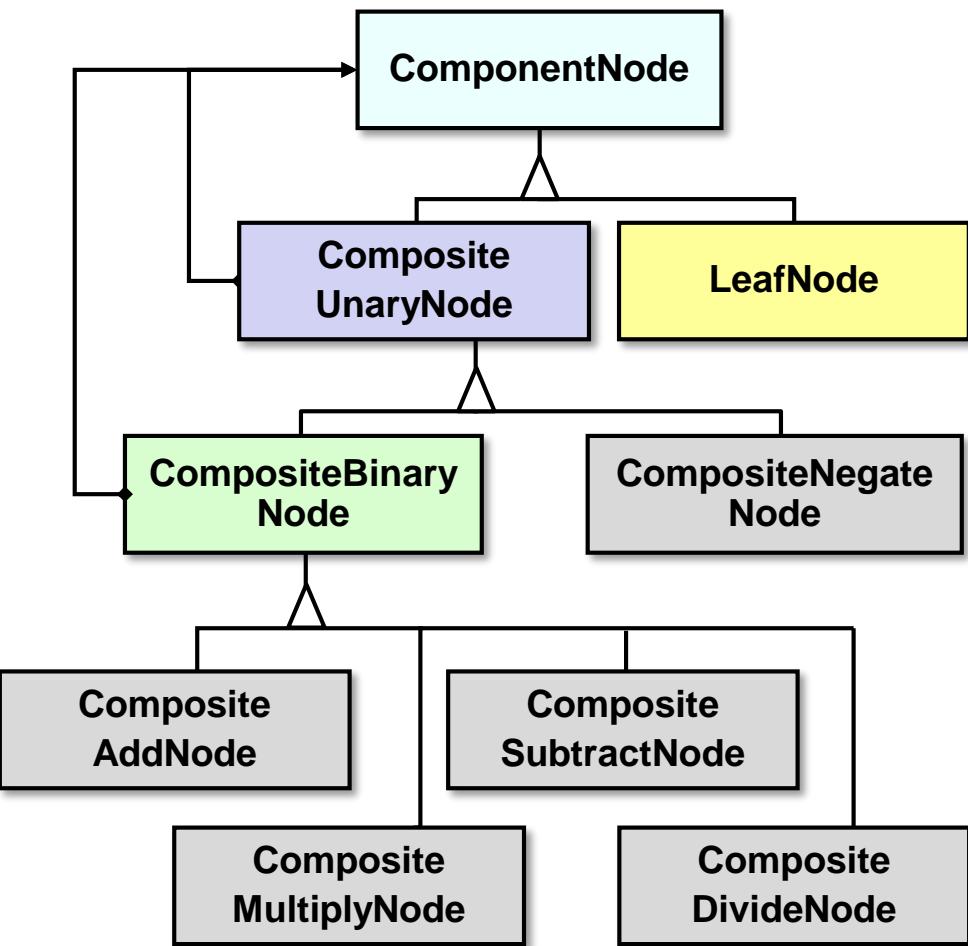
Consequences

– Perceived complexity

- May need what seems like a prohibitively large # of classes and/or objects



versus



Algorithmic decomposition

Pattern- & OO-decomposition

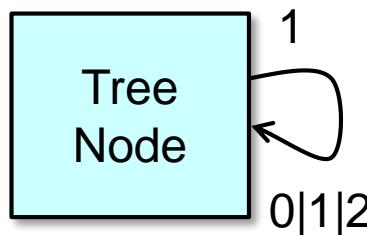
Composite

GoF Object structural

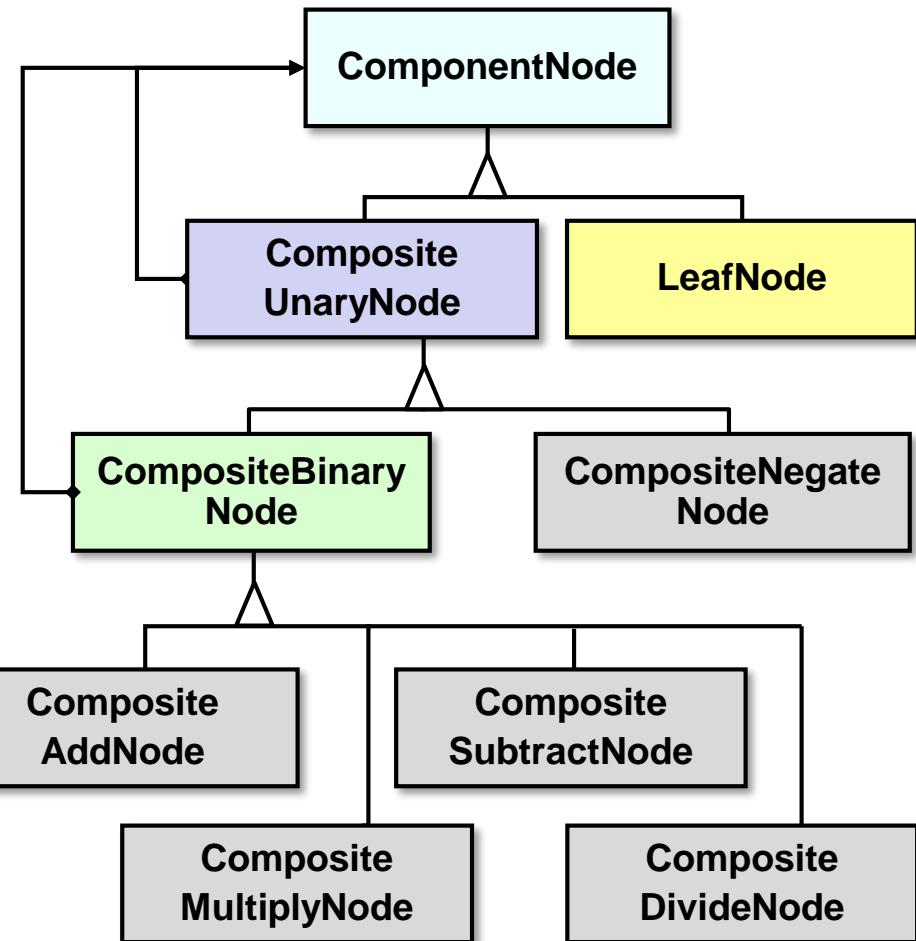
Consequences

- *Perceived complexity*

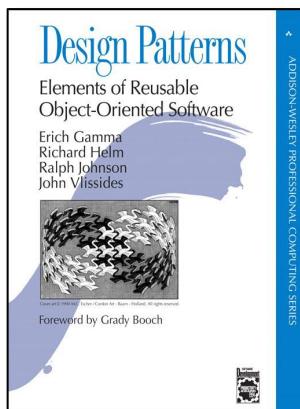
- May need what seems like a prohibitively large # of classes and/or objects



versus



Algorithmic decomposition



Pattern- & OO-decomposition

Knowledge of patterns is essential to alleviate perceived complexity

Consequences

- *Perceived complexity*
- *Awkward designs*
 - May yield “bloated” interfaces for composites & leaves

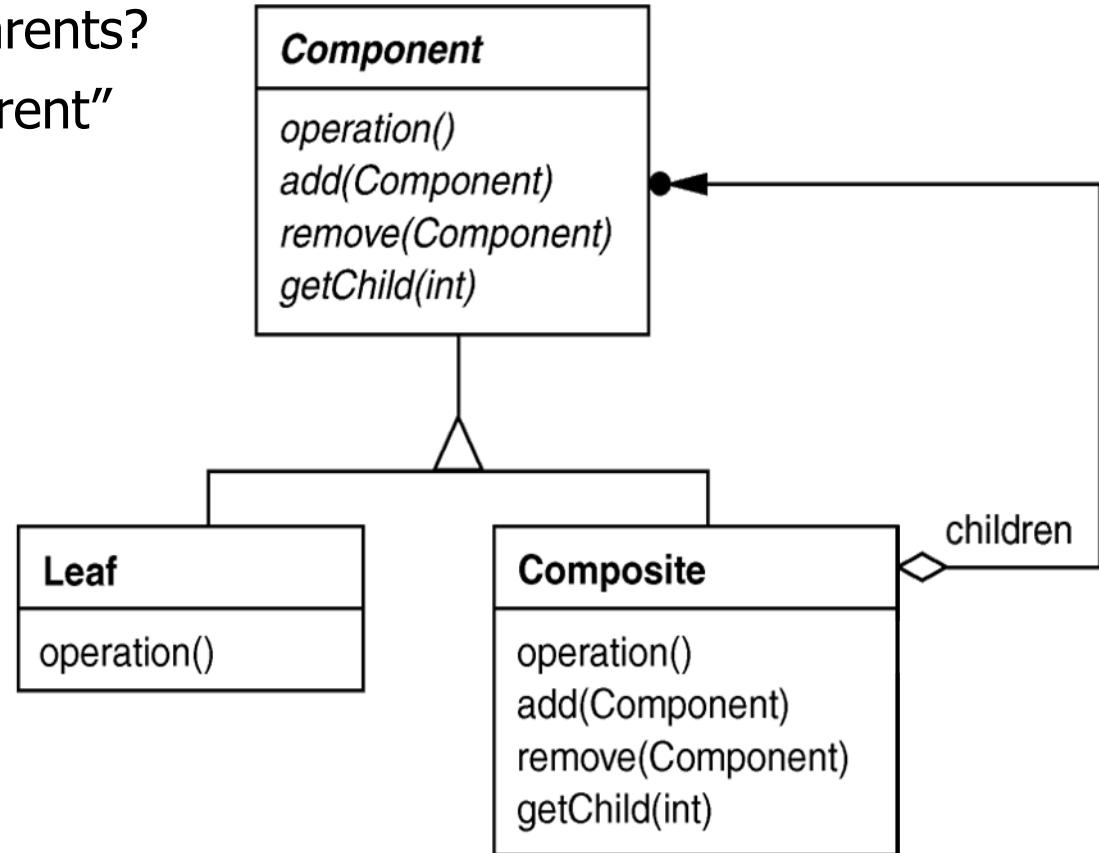
getItem() is unused in composite nodes

getLeftChild() & getRightChild() are unused in leaf nodes

int <u>ComponentNode</u> <u>ComponentNode</u> void	<u>getItem()</u> <u>getLeftChild()</u> <u>getRightChild()</u> <u>accept(Visitor visitor)</u>
---	---

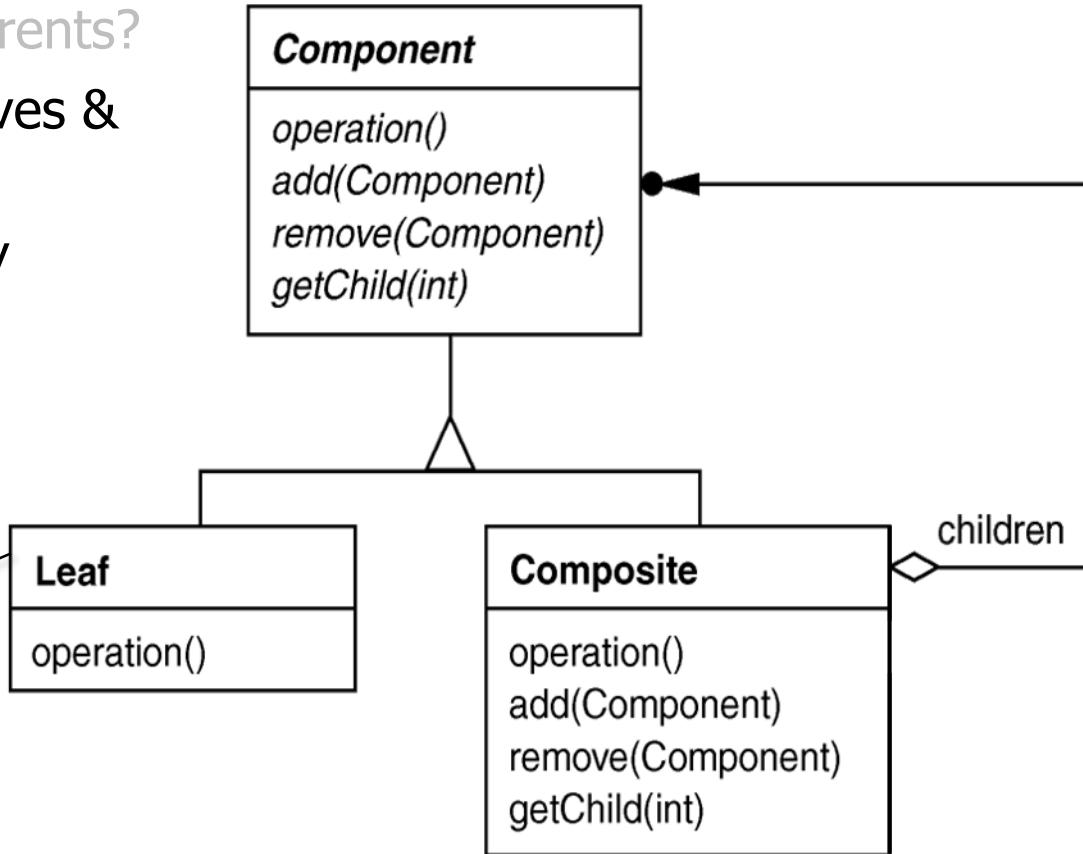
Implementation considerations

- Do components know their parents?
 - e.g., is there an explicit “parent” pointer/reference?



Implementation considerations

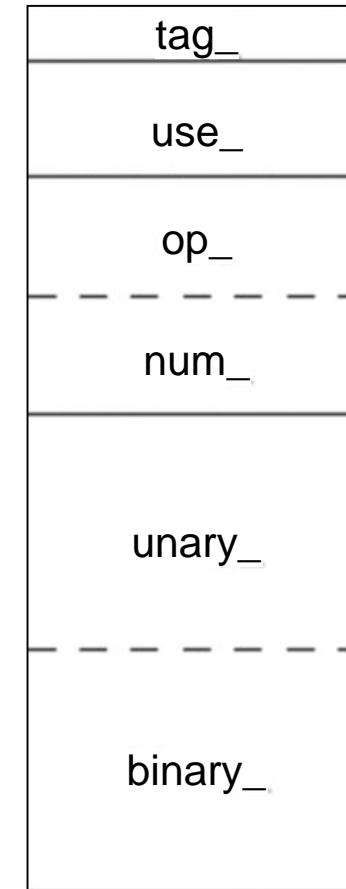
- Do components know their parents?
- Uniform interface for both leaves & composites?
 - Tradeoff between uniformity & parsimony



Implementation considerations

- Do components know their parents?
- Uniform interface for both leaves & composites?
- Don't allocate child storage in component superclass

```
typedef struct TreeNode {  
    enum { NUM, UNARY, BINARY } tag_;  
    short use_;  
    union {  
        char op_[3]; int num_;  
    } o_;  
    union {  
        struct TreeNode *unary_;  
        struct { struct TreeNode *l_,  
                 *r_; } binary_;  
    } c_;  
} TreeNode;
```

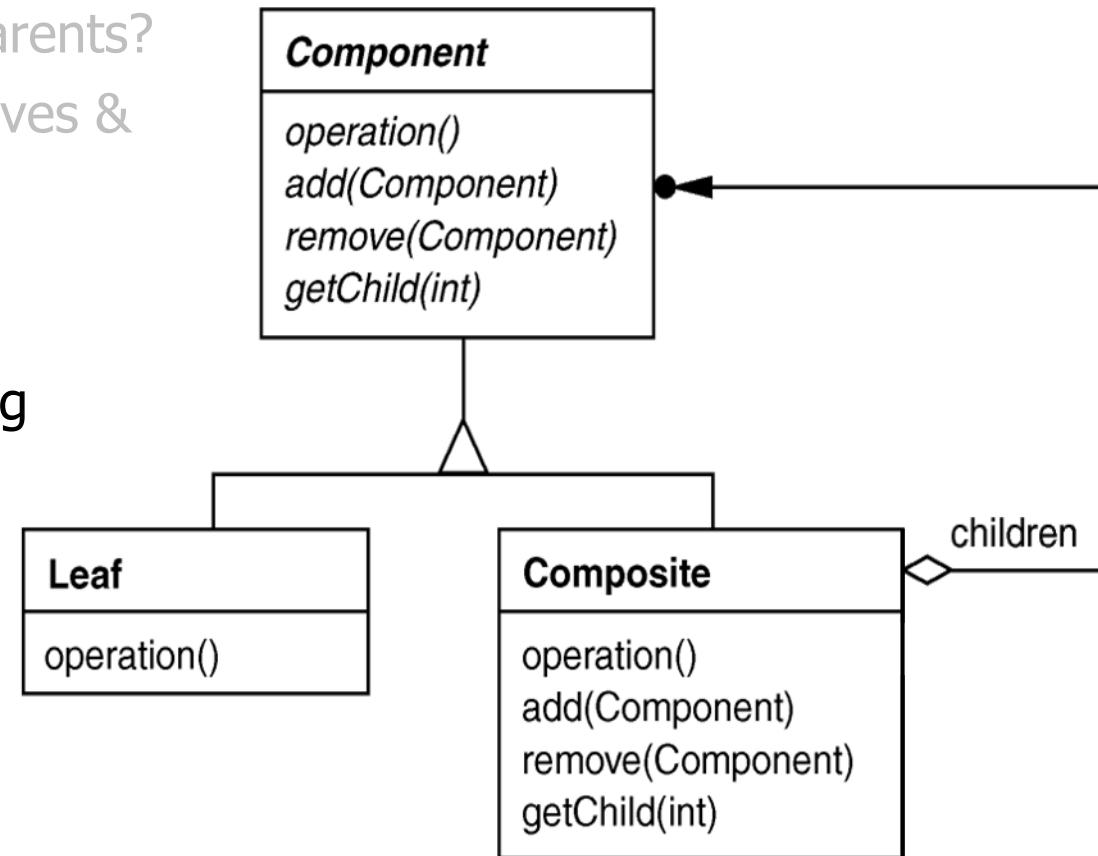


Memory
Layout

*This was a big problem with
the algorithmic decomposition*

Implementation considerations

- Do components know their parents?
- Uniform interface for both leaves & composites?
- Don't allocate child storage in component superclass
- Who is responsible for deleting children?
 - e.g., the parent or the child itself?



See rmdir vs. /bin/rm -rf at www.linfo.org/rmdir.html

Known uses

- ET++ Vobjects
- InterViews Glyphs, Styles
- Unidraw Components, MacroCommands
- Internal representations of MIME types
- Directory structures on UNIX & Windows
- `java.awt.Container #add(Component)`
- Naming Contexts in CORBA

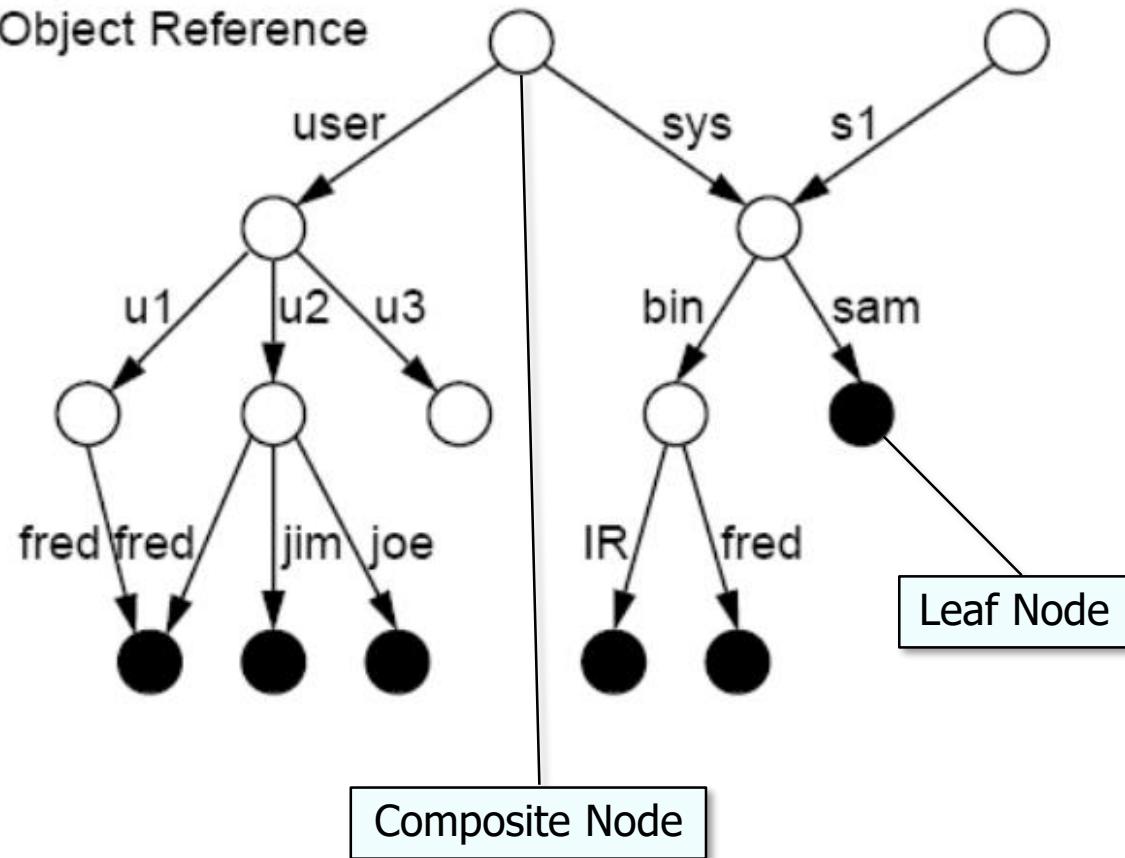
○ = Context

● = Object

→ = Object Reference

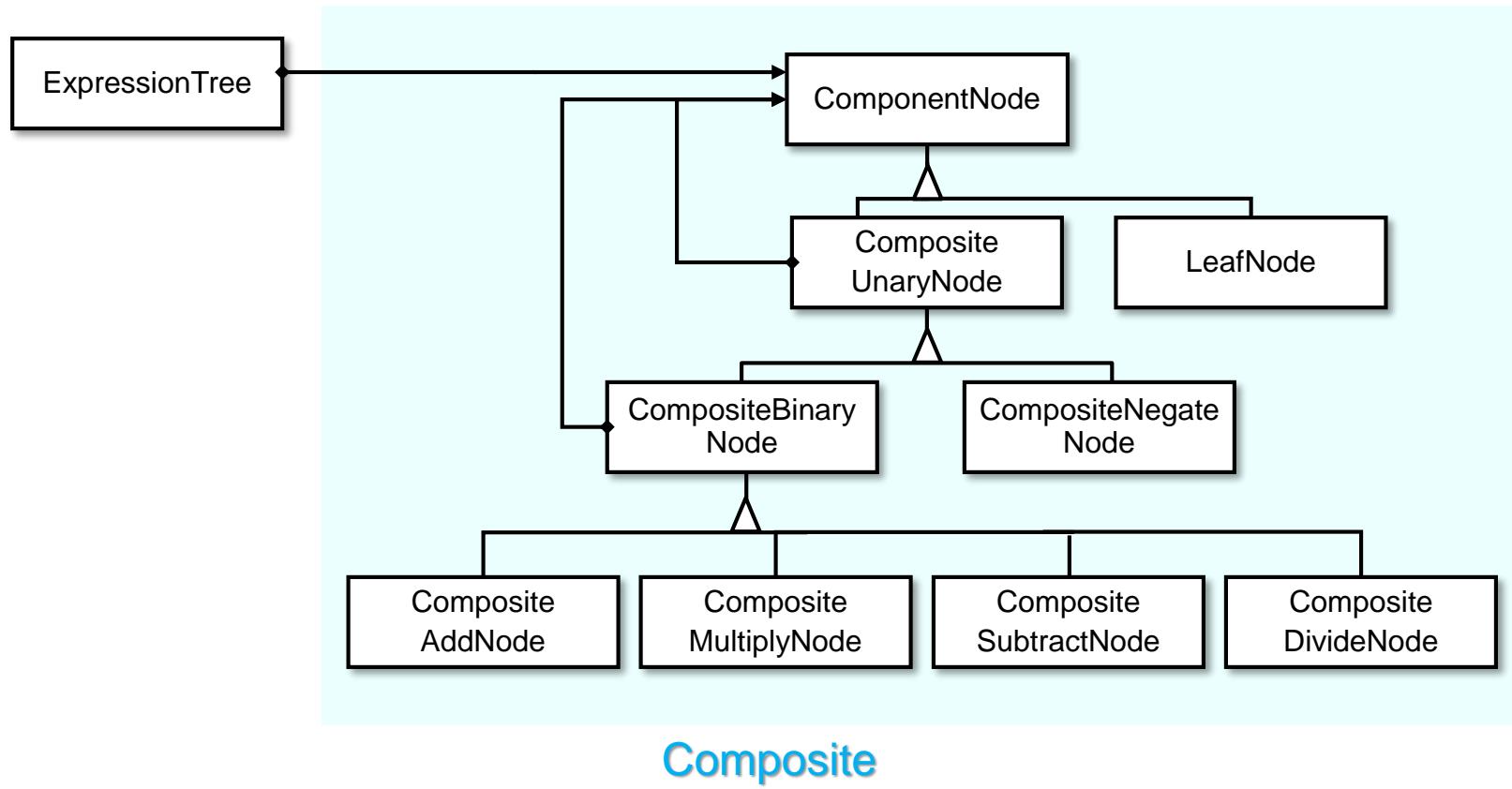
Orphaned Context

Orphaned Context



Summary of the Composite Pattern

- The expression tree processing app uses the *Composite* pattern to enhance the uniformity & extensibility of its key internal data structure



Adding new types of nodes (& new operations on nodes) is greatly simplified

End of the
Composite Pattern

The Bridge Pattern

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software
Integrated Systems

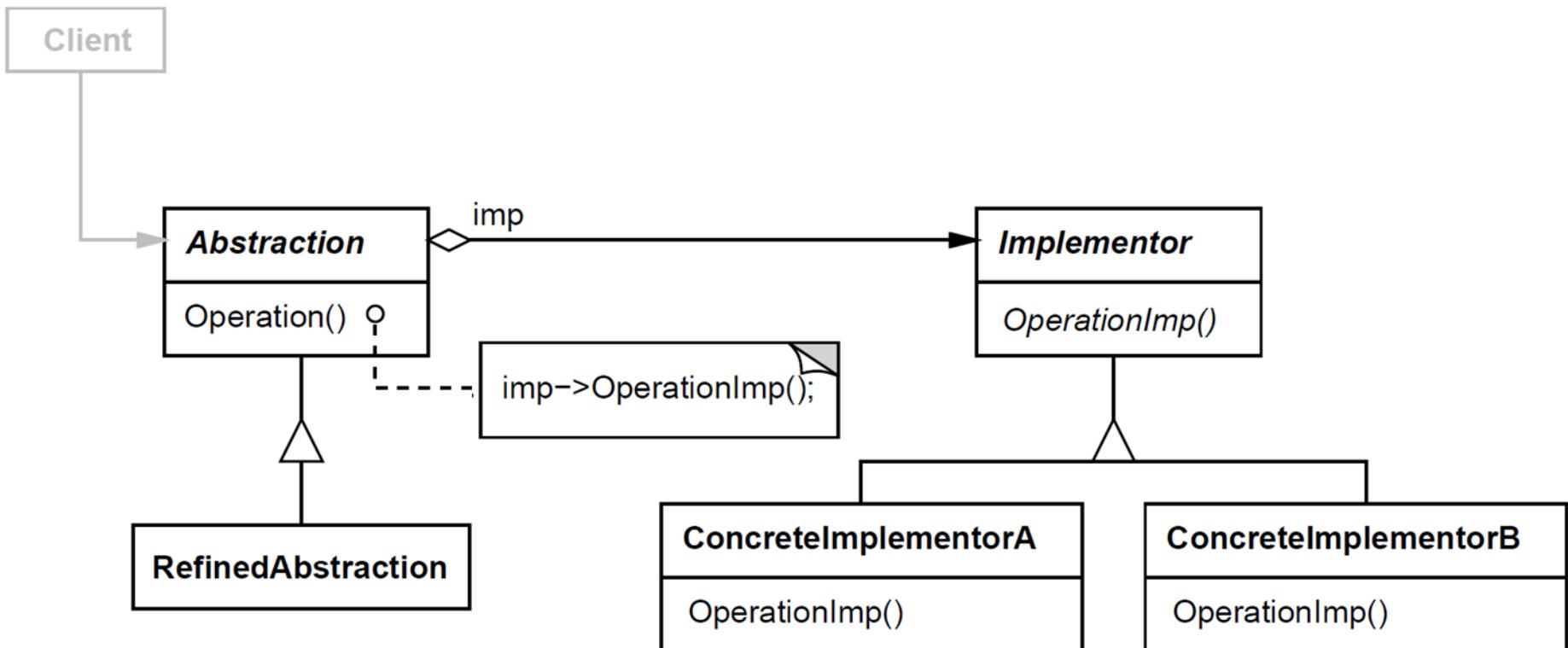
Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives

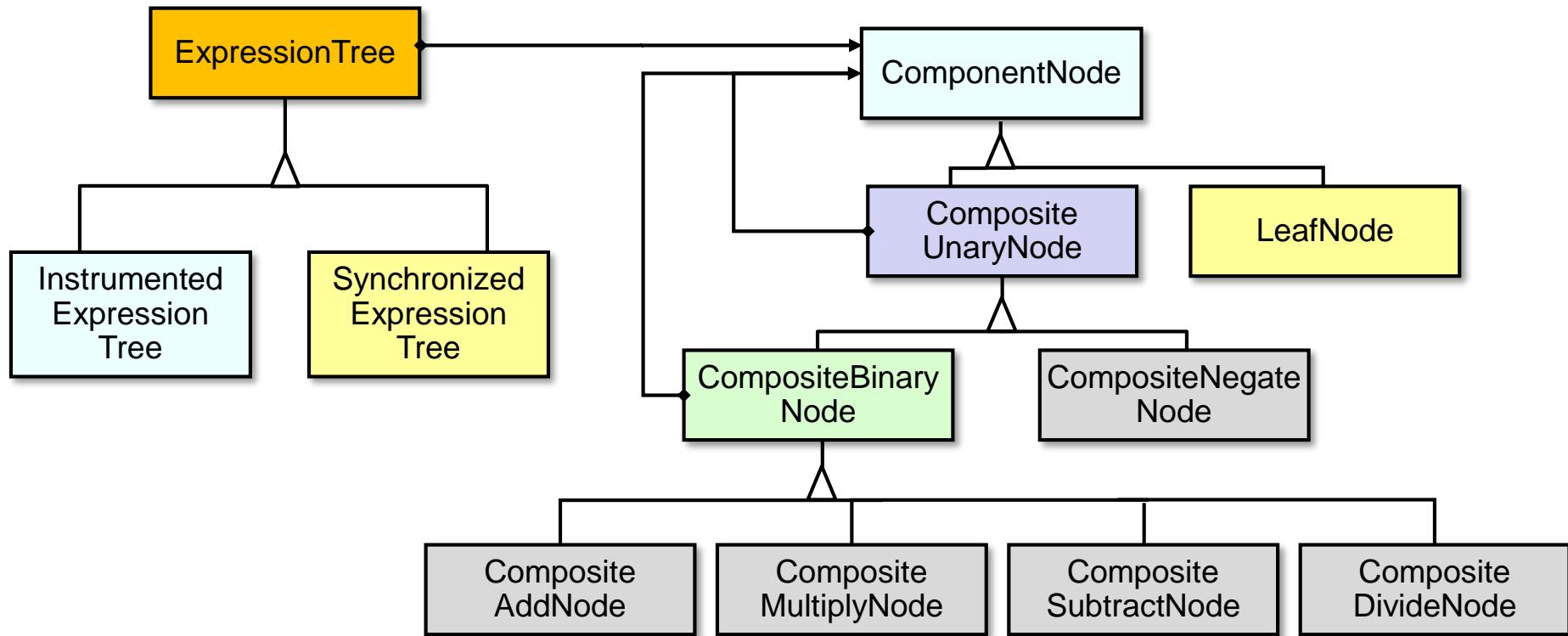
Learning Objectives

- Understand the *Bridge* pattern



Learning Objectives

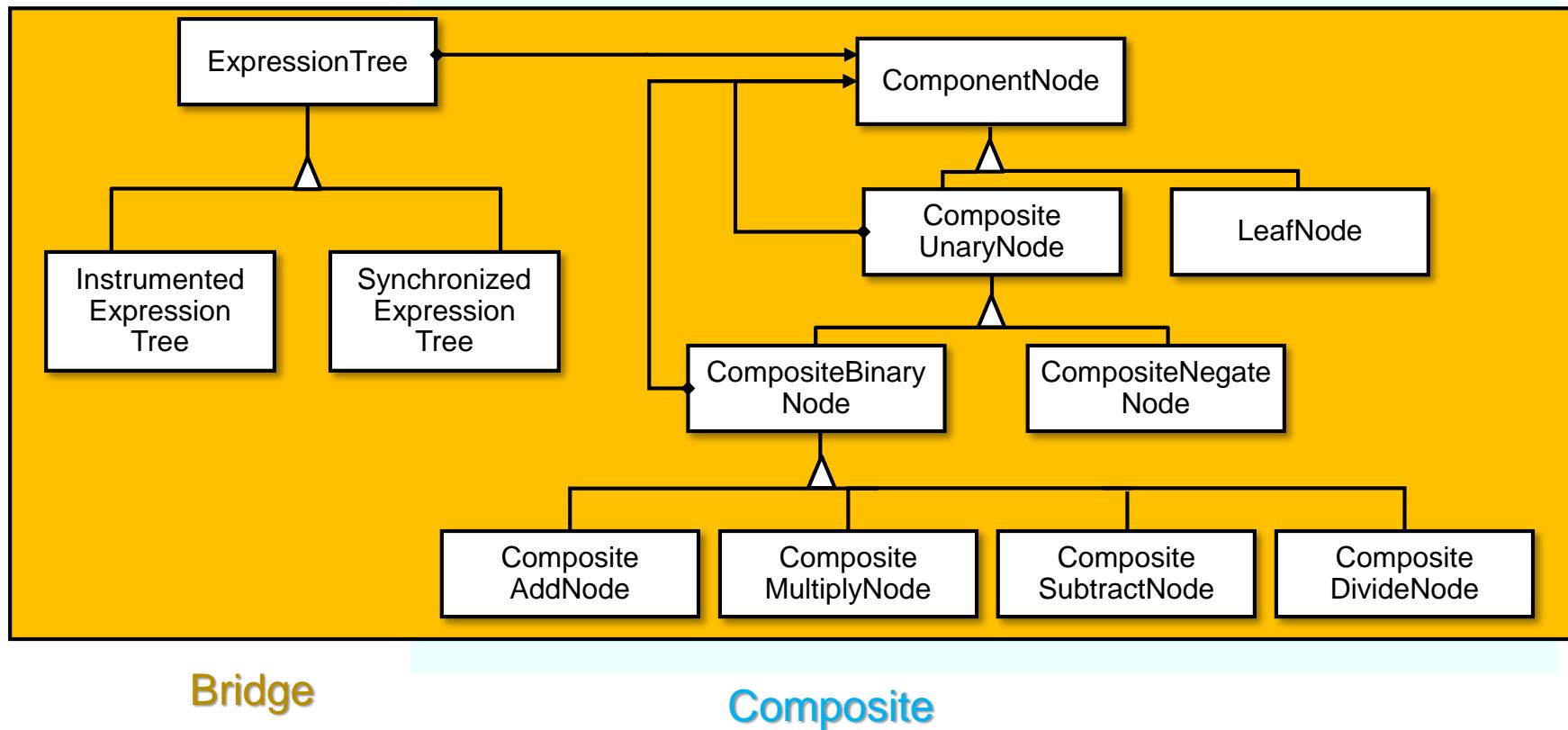
- Understand the *Bridge* pattern
- Recognize how *Bridge* can be applied to make the expression tree structure easier to access & evolve transparently



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

A Pattern for Binding One of Many Variations

Purpose: Decouple expression tree programming API from its behavior & implementation & enable transparent extensibility



Bridge minimizes coupling between clients, abstractions, & implementations

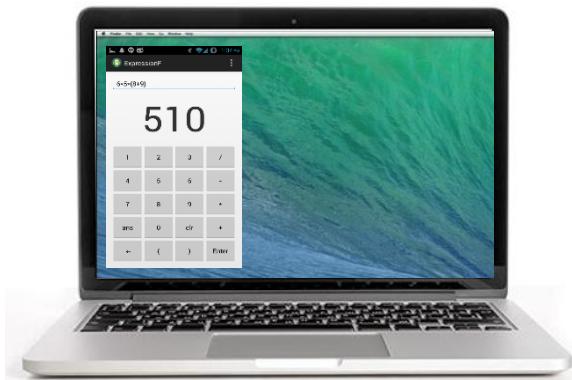
Context: OO Expression Tree Processing App

- The app needs to run in a range of design-time & runtime environments



Context: OO Expression Tree Processing App

- The app needs to run in a range of design-time & runtime environments, e.g.
 - Mobile devices with limited memory & processing power



Context: OO Expression Tree Processing App

- The app needs to run in a range of design-time & runtime environments, e.g.
 - Mobile devices with limited memory & processing power
 - Laptops & desktops with more abundance resources



Problem: Minimizing Impact of Variability

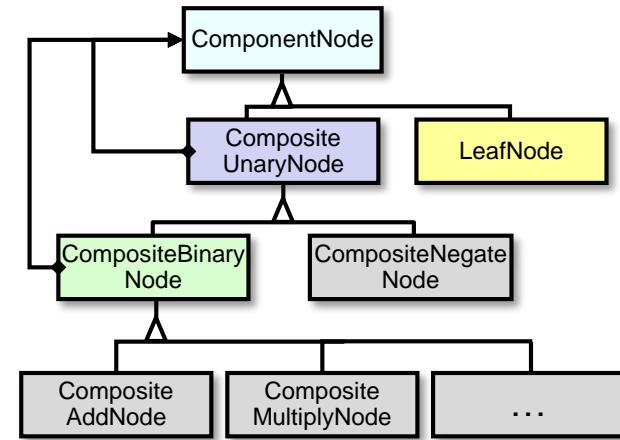
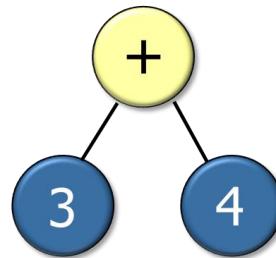
- Tightly coupling app components to a particular environment has drawbacks



Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks
 - Sub-optimal implementations for a given context

```
ComponentNode node =  
    new CompositeAddNode  
    (new LeafNode(3),  
     new LeafNode(4));
```



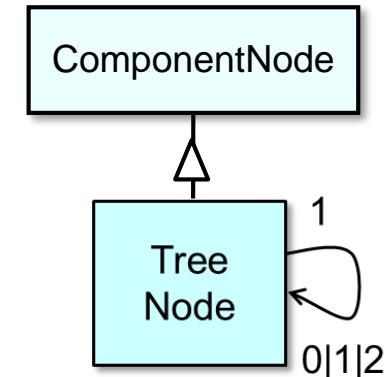
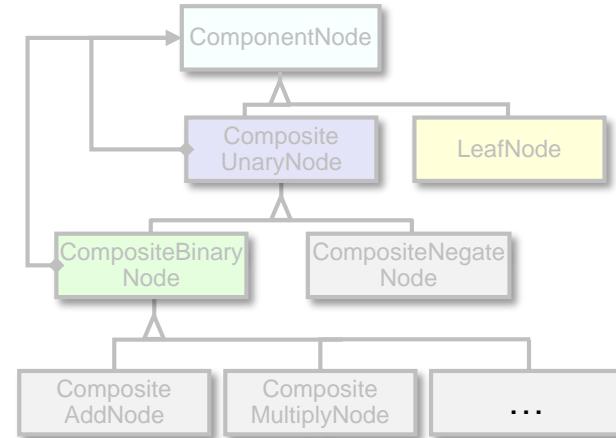
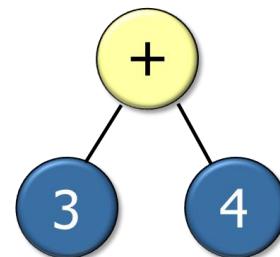
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks
 - Sub-optimal implementations for a given context

```
ComponentNode node =  
    new CompositeAddNode  
    (new LeafNode(3),  
     new LeafNode(4));
```

versus

```
ComponentNode node =  
    new TreeNode  
    ('+',  
     new TreeNode(3),  
     new TreeNode(4));
```



Problem: Minimizing Impact of Variability

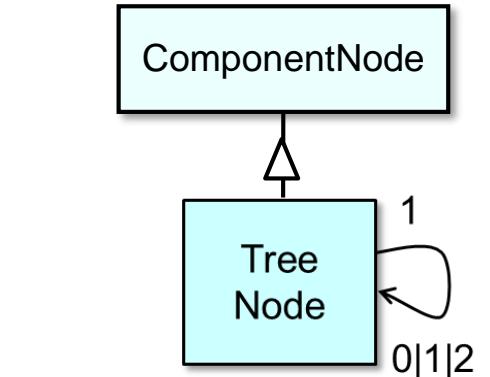
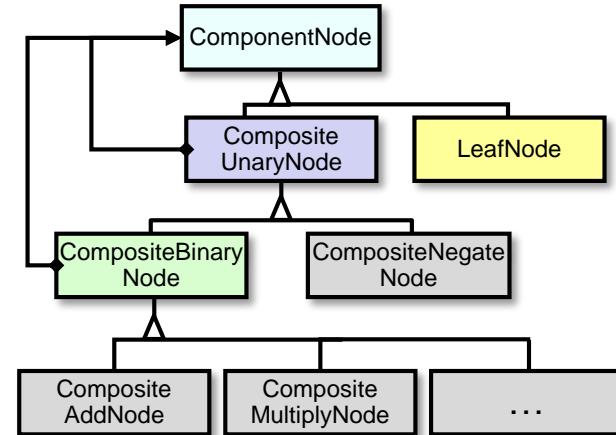
- Tightly coupling app components to a particular environment has drawbacks
 - Sub-optimal implementations for a given context

```
ComponentNode node =  
    new CompositeAddNode  
    (new LeafNode(3),  
     new LeafNode(4));
```



Different implementations have different time/space tradeoffs

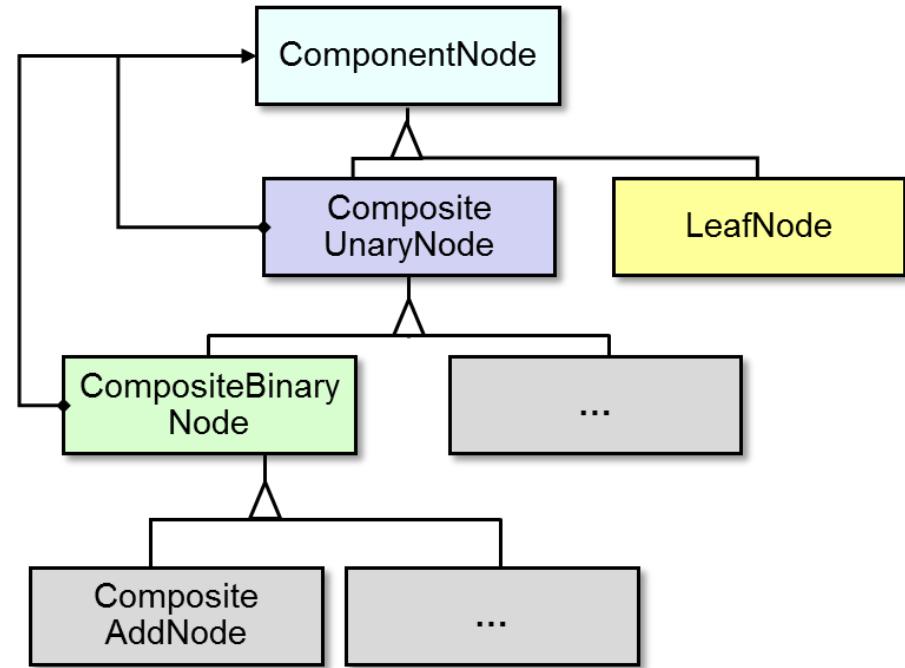
```
ComponentNode node =  
    new TreeNode  
    ('+',  
     new TreeNode(3),  
     new TreeNode(4));
```



We should be able to change implementations without breaking client code

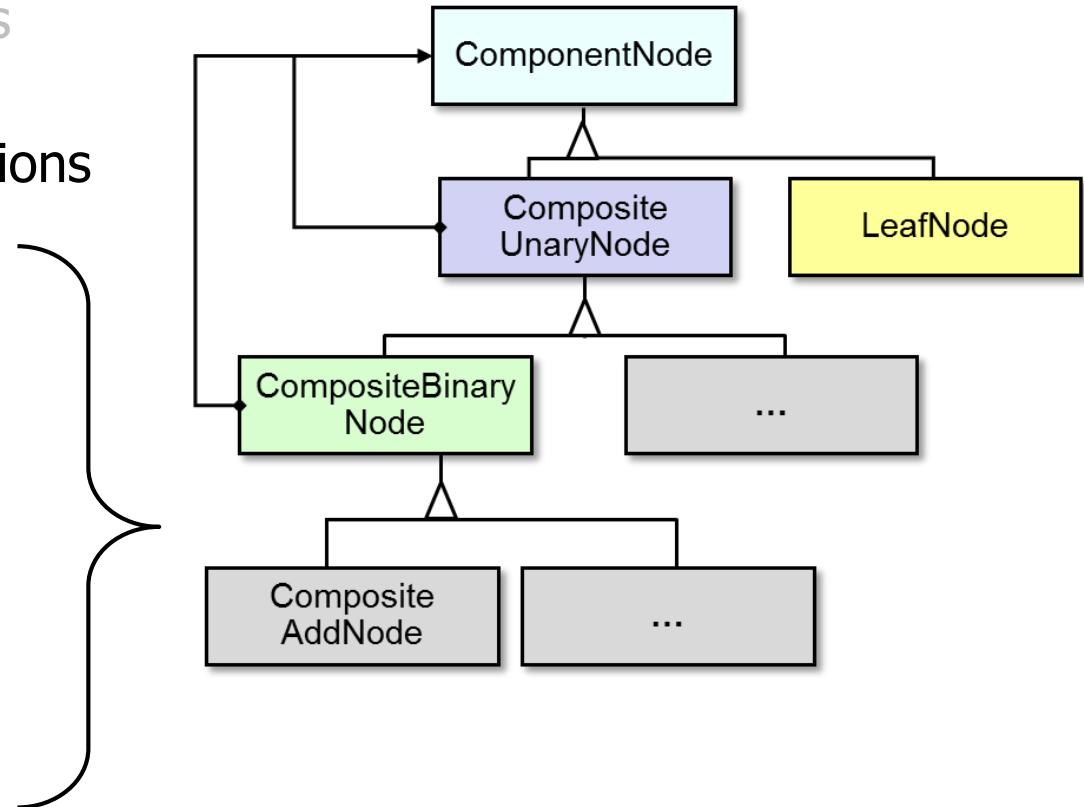
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks
 - Sub-optimal implementations for a given context
 - Hard to change implementations transparently



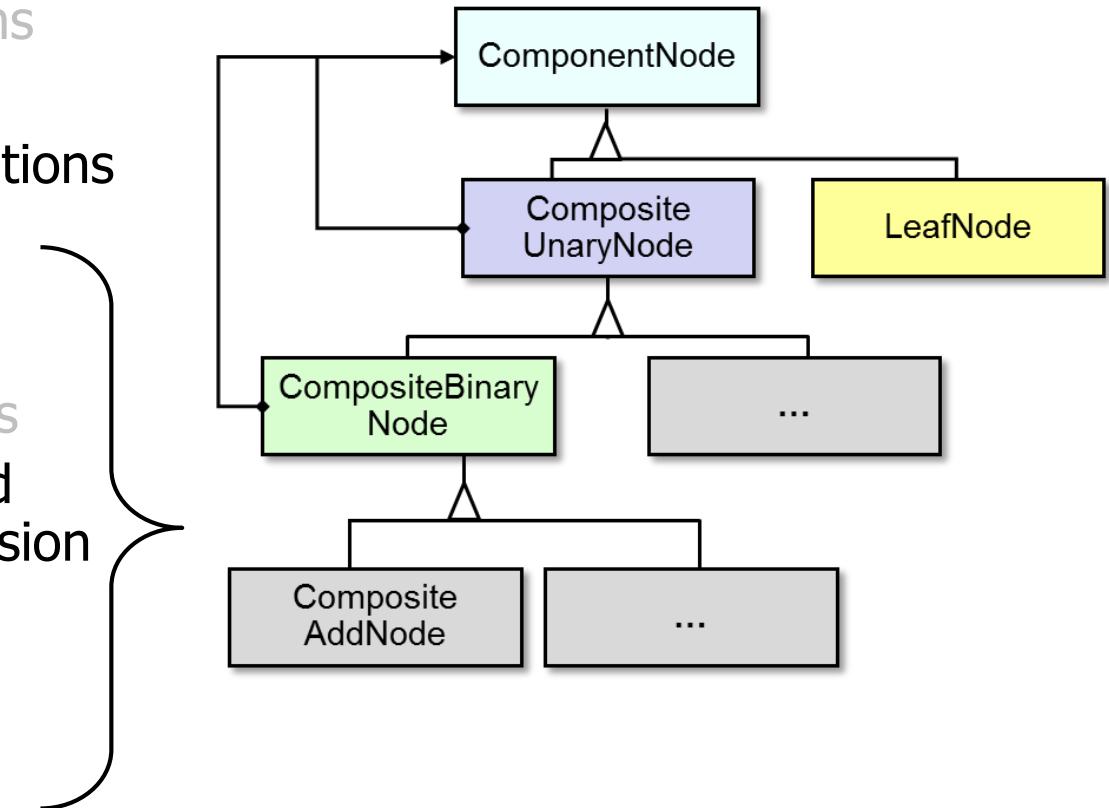
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks
 - Sub-optimal implementations for a given context
 - Hard to change implementations transparently, e.g.
 - Want to transparently add instrumentation to expression tree operations



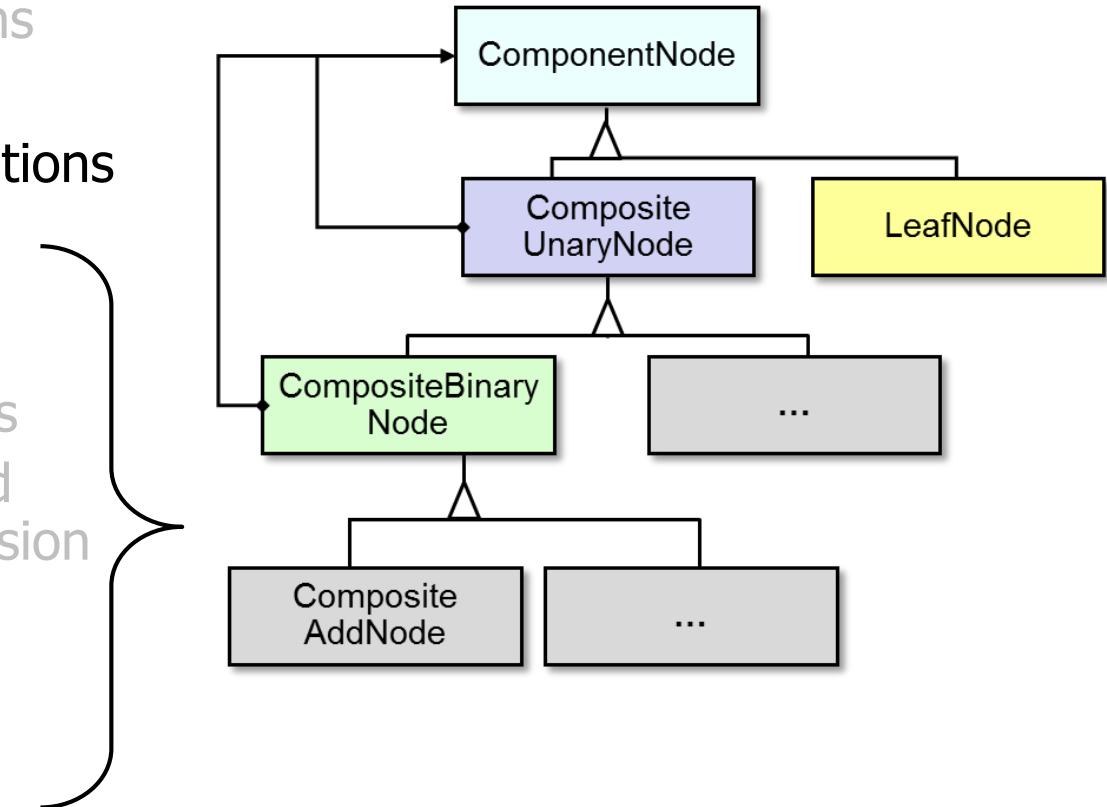
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks
 - Sub-optimal implementations for a given context
 - Hard to change implementations transparently, e.g.
 - Want to transparently add instrumentation to expression tree operations
 - Want to transparently add synchronization to expression tree methods



Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks
 - Sub-optimal implementations for a given context
 - Hard to change implementations transparently, e.g.
 - Want to transparently add instrumentation to expression tree operations
 - Want to transparently add synchronization to expression tree methods
 - ...



We should be able to enhance the service without breaking the implementation

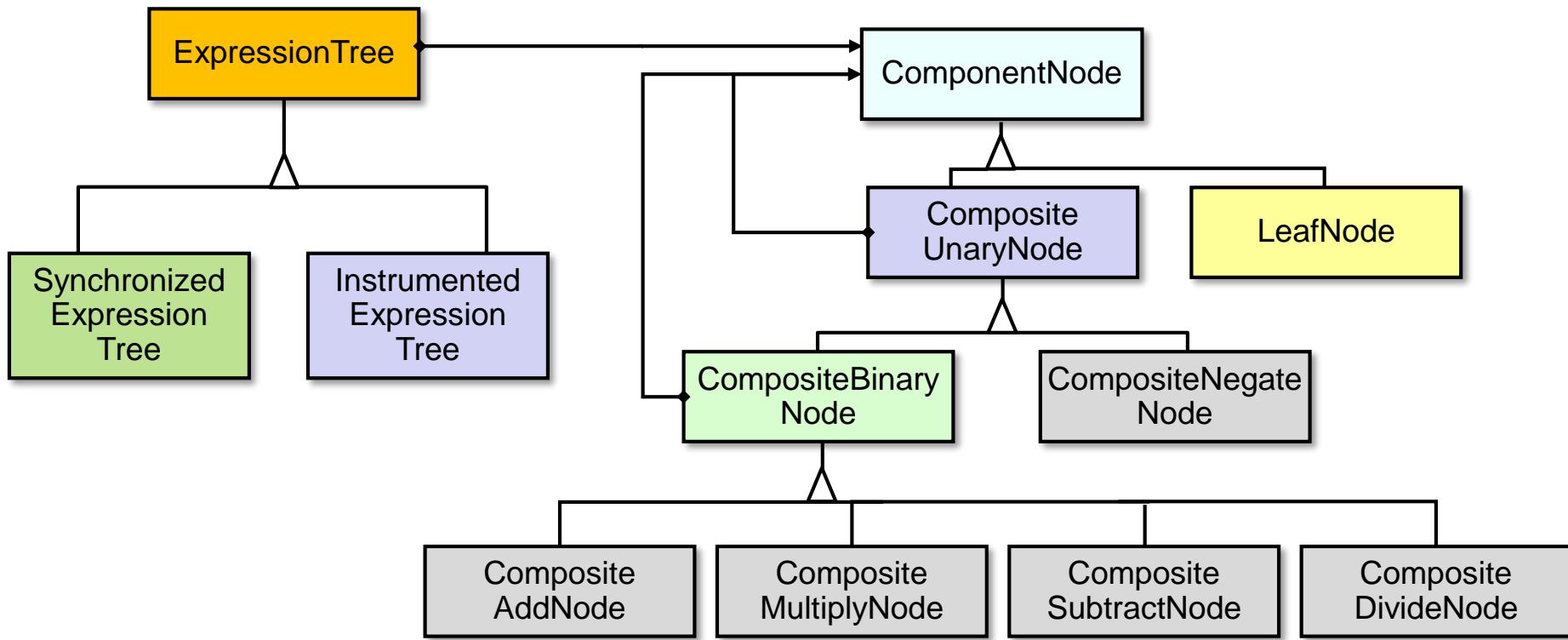
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction

ExpressionTree

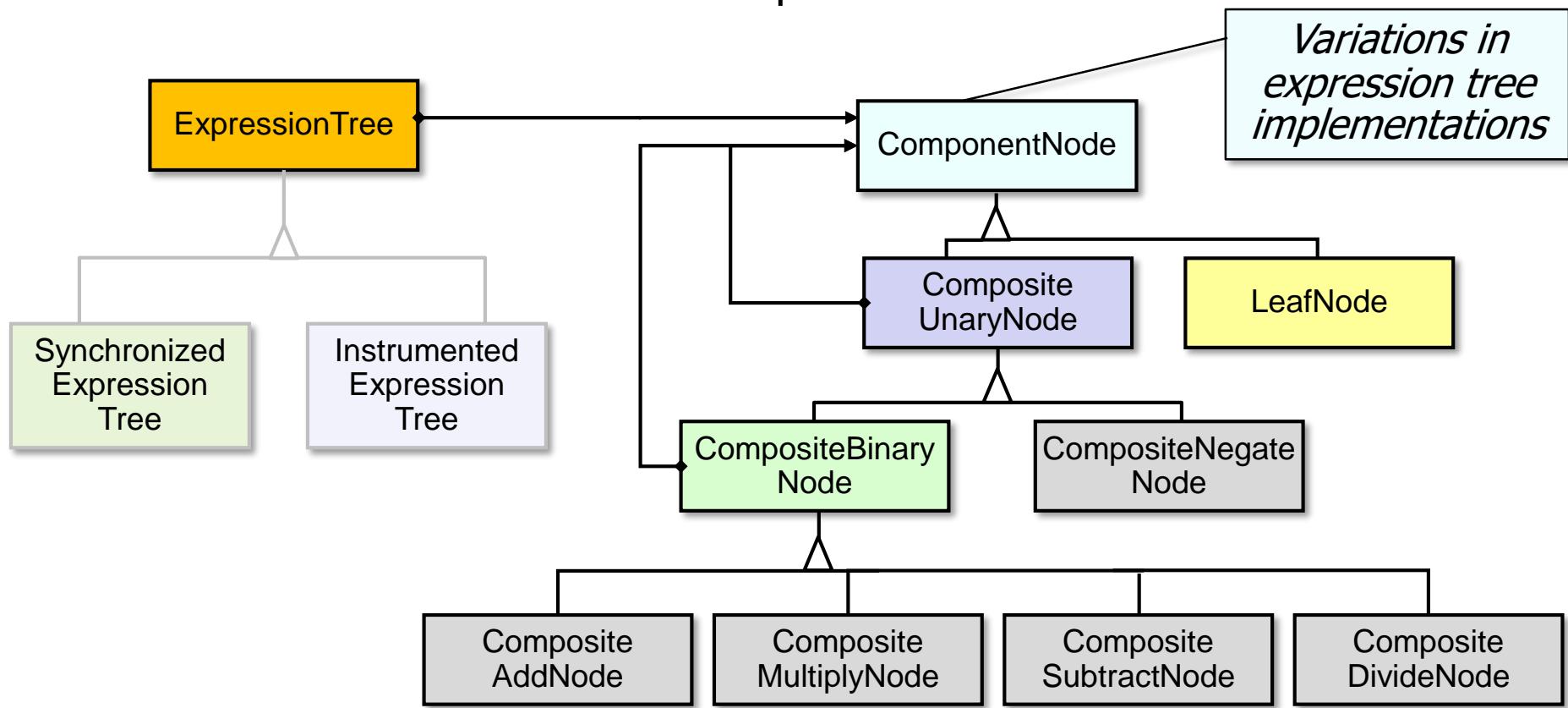
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations



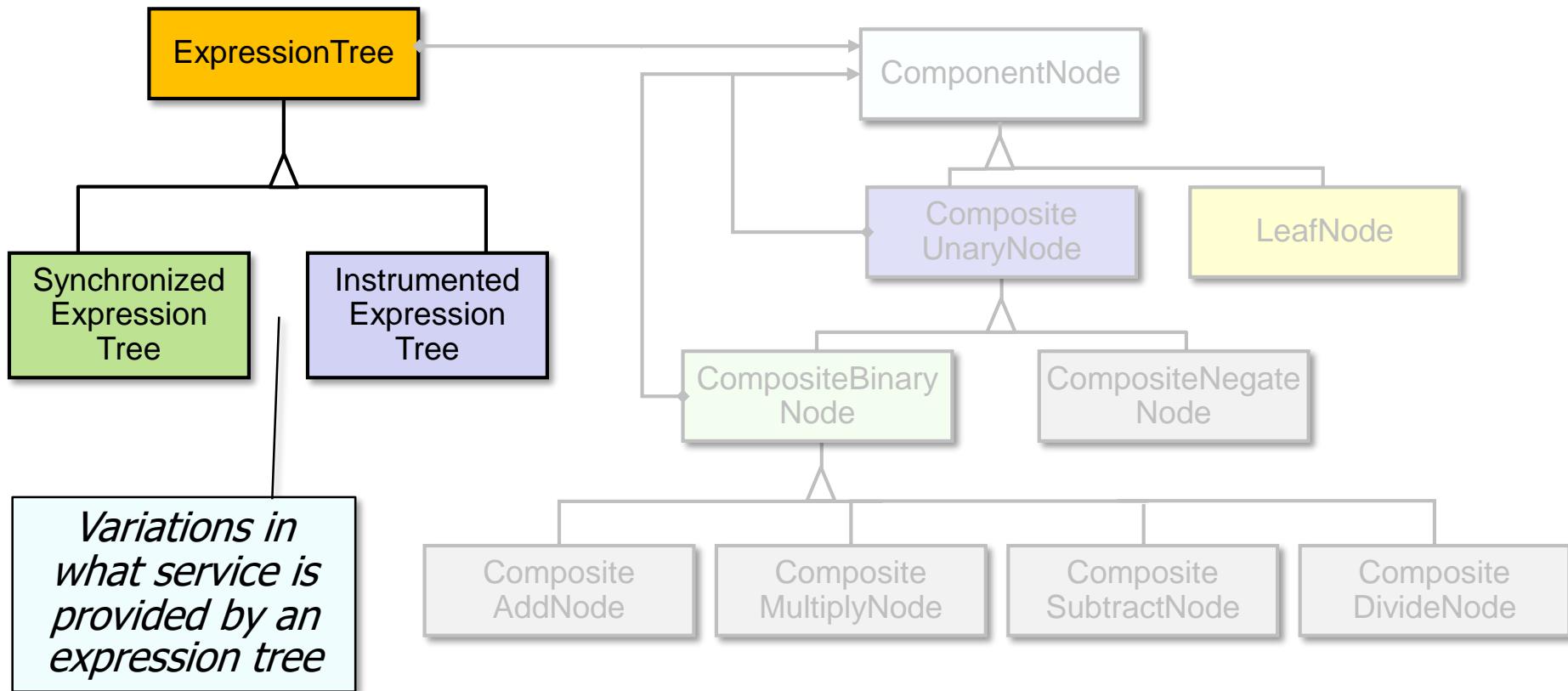
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations



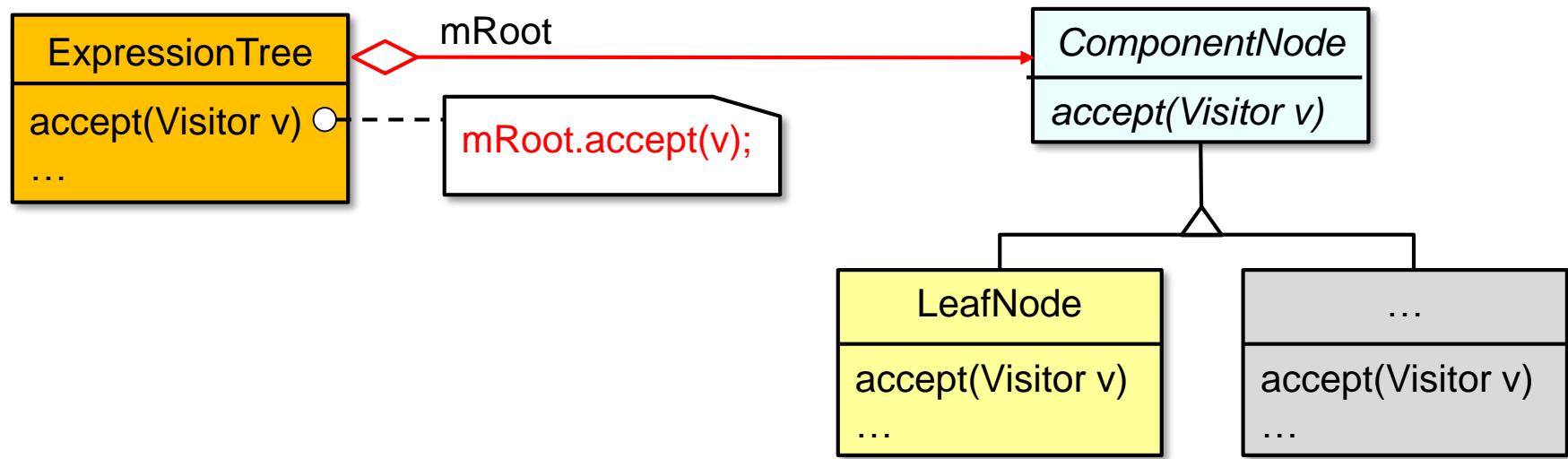
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations



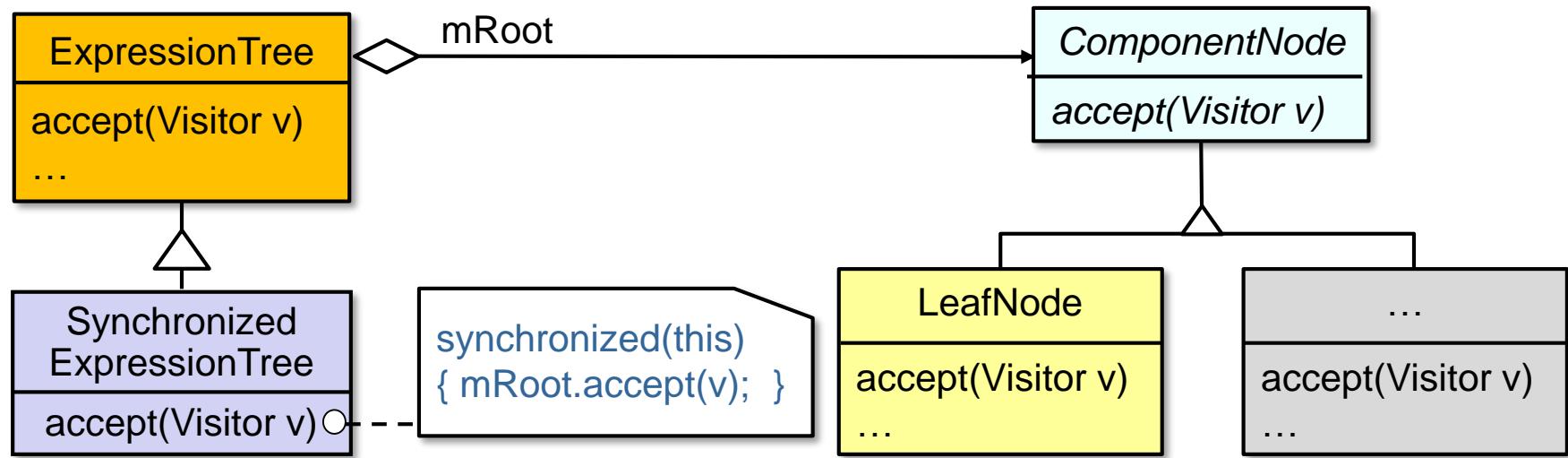
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations
 - Client calls to the abstraction are forwarded to the corresponding implementor subclass



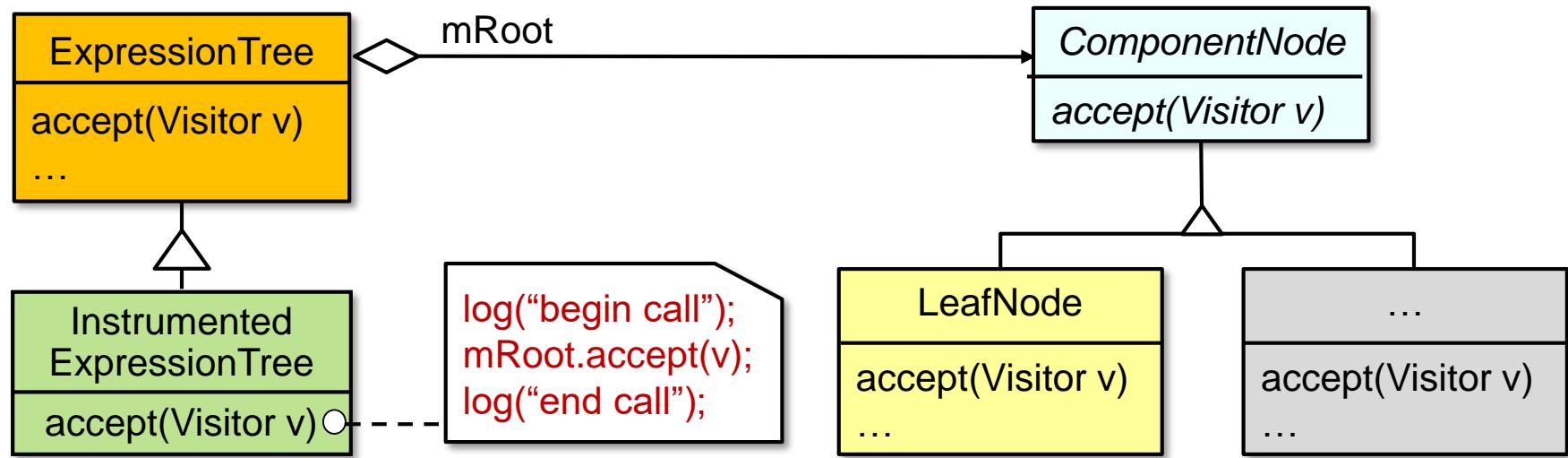
Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations
 - Client calls to the abstraction are forwarded to the corresponding implementor subclass
 - Subclass the abstraction class to enable different services without affecting the implementor hierarchy



Solution: Separate Abstraction & Implementation

- Encapsulate variability behind a stable API that creates separate class hierarchies for an abstraction & its implementations
 - Client calls to the abstraction are forwarded to the corresponding implementor subclass
 - Subclass the abstraction class to enable different services without affecting the implementor hierarchy



ExpressionTree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

Class methods

```
    ExpressionTree (ComponentNode root)
    boolean isNull ()
    int getItem ()
    ExpressionTree getLeftChild ()
    ExpressionTree getRightChild ()
    void accept (Visitor visitor)
    Iterator
<ExpressionTree> iterator (String traversalOrder)
```

ExpressionTree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

Class methods

Pass in the root of the implementor hierarchy 

```
ExpressionTree(ComponentNode root)
boolean isNull()
int getItem()
ExpressionTree getLeftChild()
ExpressionTree getRightChild()
void accept(Visitor visitor)
Iterator
<ExpressionTree> iterator(String traversalOrder)
```

ExpressionTree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

Class methods

Forward to
implementor
hierarchy



```
ExpressionTree (ComponentNode root)
    boolean isNullgetItem()
    ExpressionTree getLeftChildExpressionTree getRightChildaccept (Visitor visitor)
    Iterator
<ExpressionTree> iterator (String traversalOrder)
```

ExpressionTree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

Class methods

```
    ExpressionTree (ComponentNode root)
boolean isNull ()
    int getItem ()
ExpressionTree getLeftChild ()
ExpressionTree getRightChild ()
    void accept (Visitor visitor) 
Iterator
<ExpressionTree> iterator (String traversalOrder)
```

Plays essential role in *Iterator* & *Visitor* patterns (covered later)

ExpressionTree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

Class methods

```
    ExpressionTree (ComponentNode root)  
    boolean isNull()  
    int getItem()  
    ExpressionTree getLeftChild()  
    ExpressionTree getRightChild()  
    void accept (Visitor visitor)  
    Iterator  
<ExpressionTree> iterator (String traversalOrder)
```



Factory method creates iterator

ExpressionTree Class Overview

- Defines an abstraction that shields clients from implementation details of expression tree that may change at design-time or runtime

Class methods

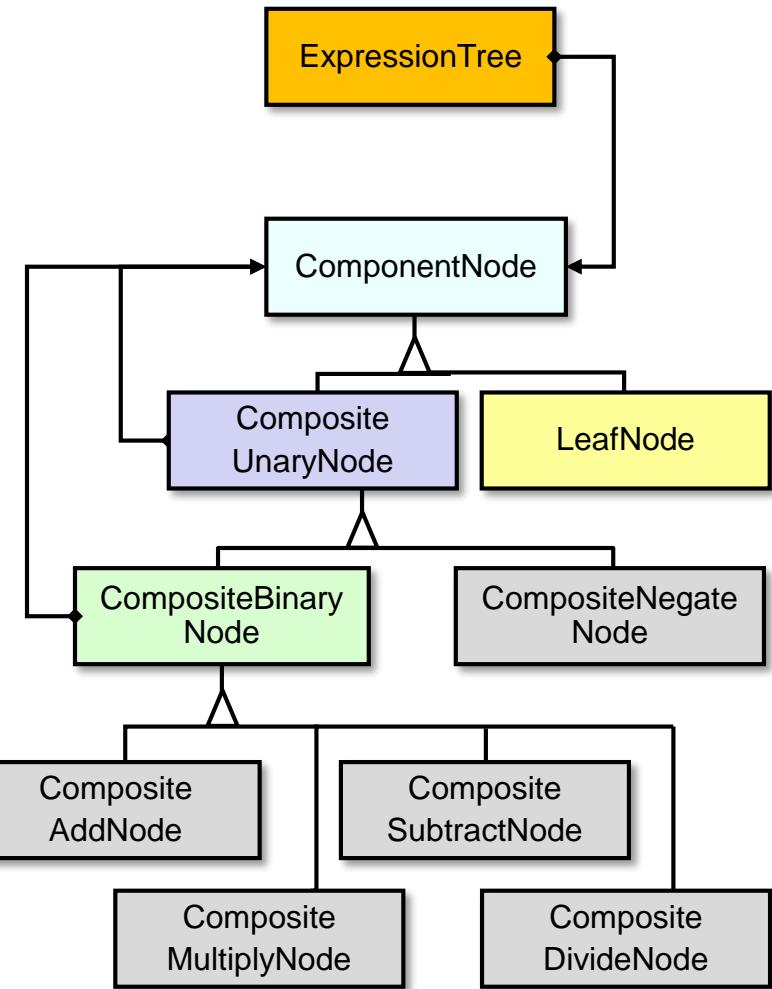
```
    ExpressionTree (ComponentNode root)
    boolean isNull ()
    int getItem ()
    ExpressionTree getLeftChild ()
    ExpressionTree getRightChild ()
    void accept (Visitor visitor)
    Iterator
<ExpressionTree> iterator (String traversalOrder)
```

- **Commonality:** Provides common interface for expression tree operations
- **Variability:** Component nodes will vary depending on user input expressions, iterator behavior can vary, & expression tree itself can vary

Elements of the Bridge Pattern

Intent

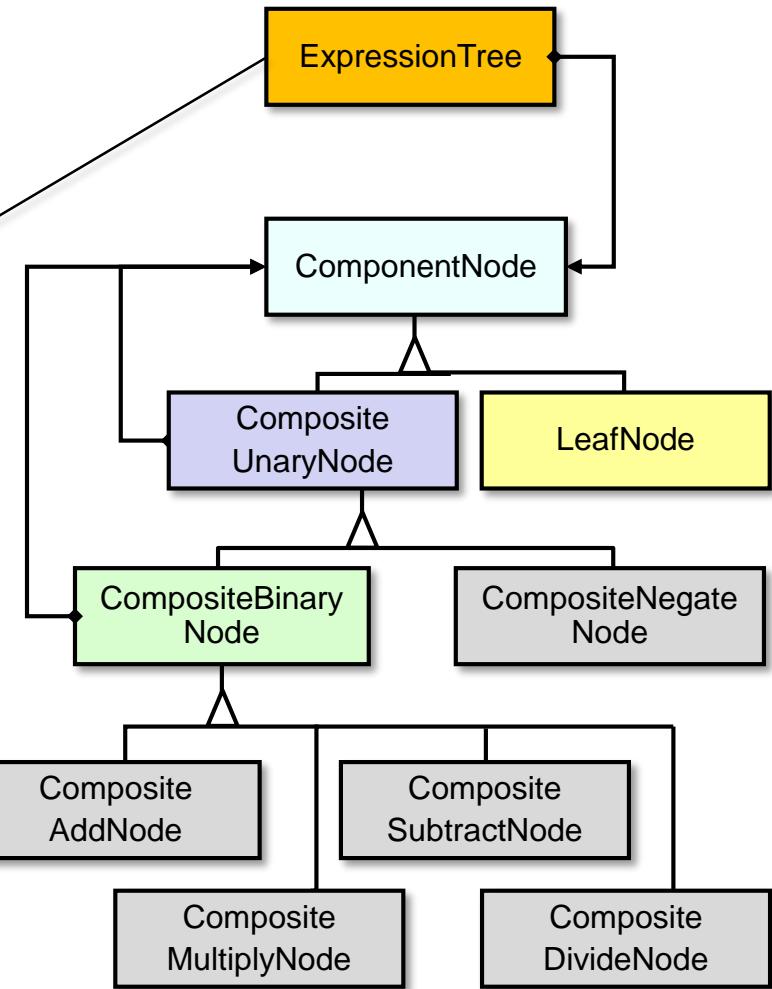
- Separate an abstraction from its implementation(s) so the two can vary independently



Applicability

- When the abstraction & extensible implementation can vary independently

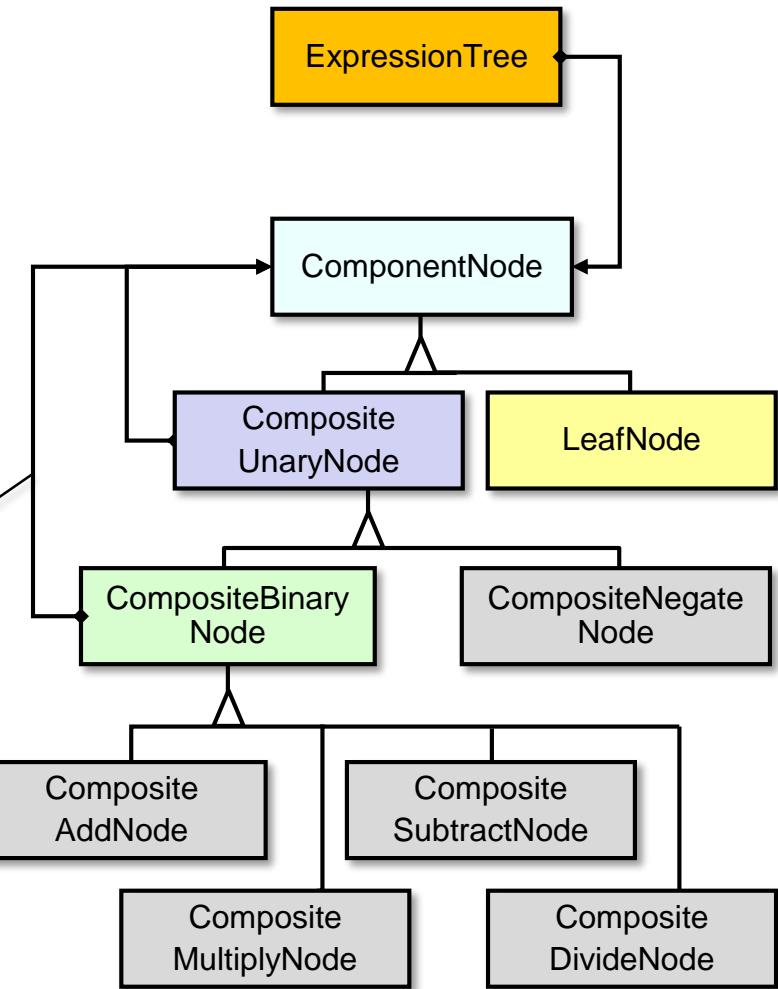
e.g., the **ExpressionTree** service can be refined without affecting clients



Applicability

- When the abstraction & extensible implementation can vary independently
- When there's a need to change implementor hierarchies at design-time or runtime without breaking client code

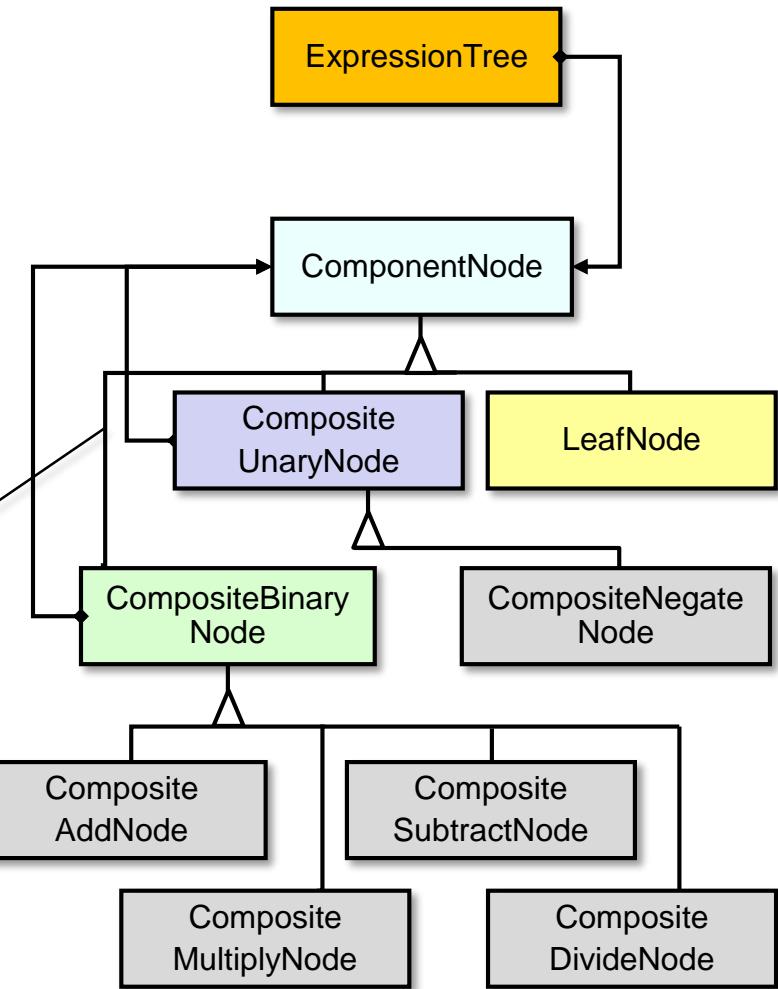
e.g., the **ComponentNode** implementor hierarchy can change without affecting clients



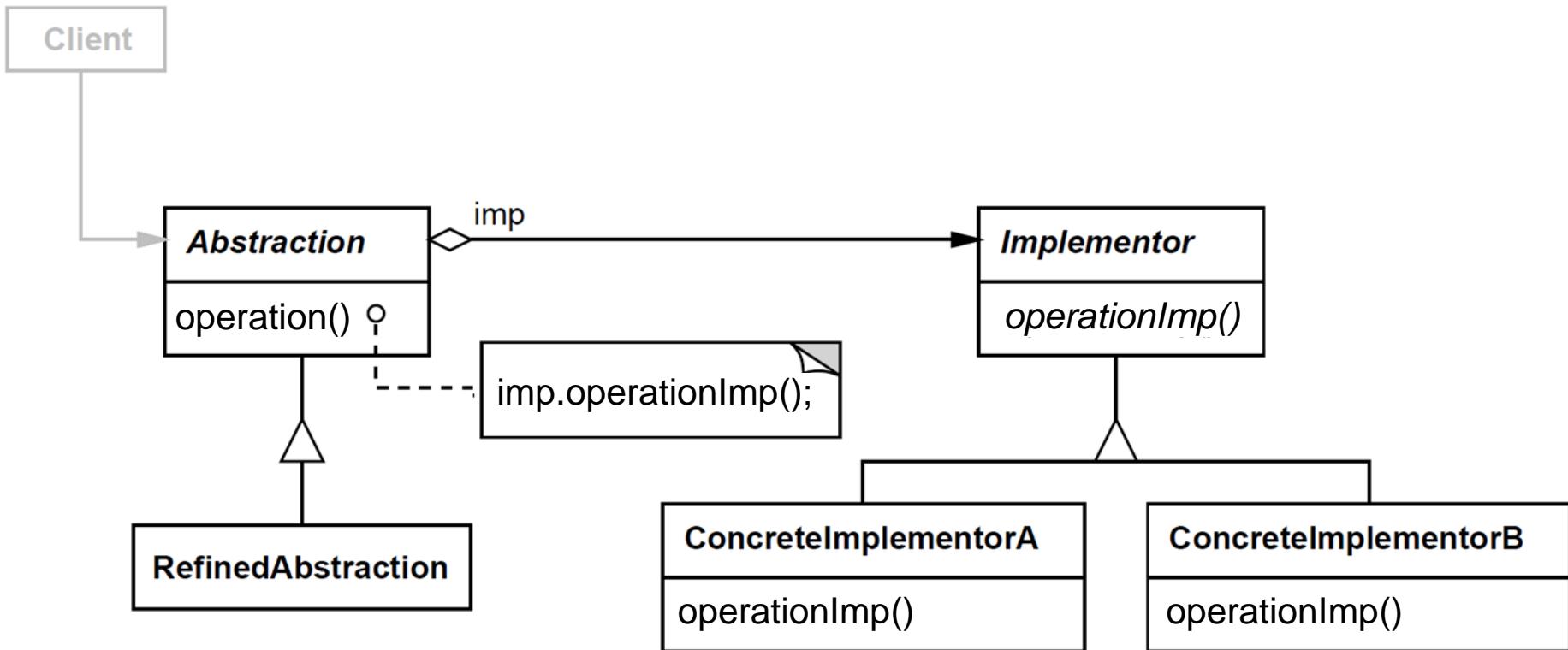
Applicability

- When the abstraction & extensible implementation can vary independently
- When there's a need to change implementor hierarchies at design-time or runtime without breaking client code

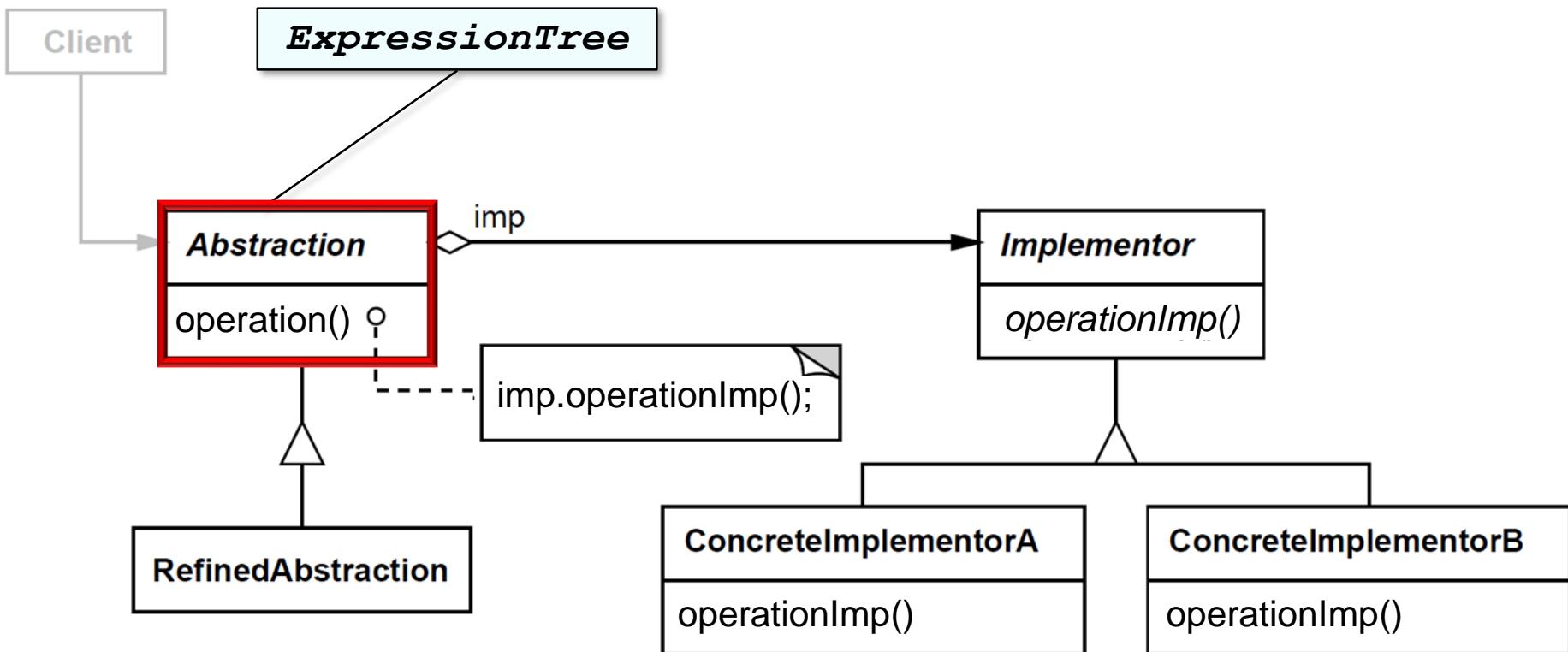
e.g., the **ComponentNode** implementor hierarchy can change without affecting clients



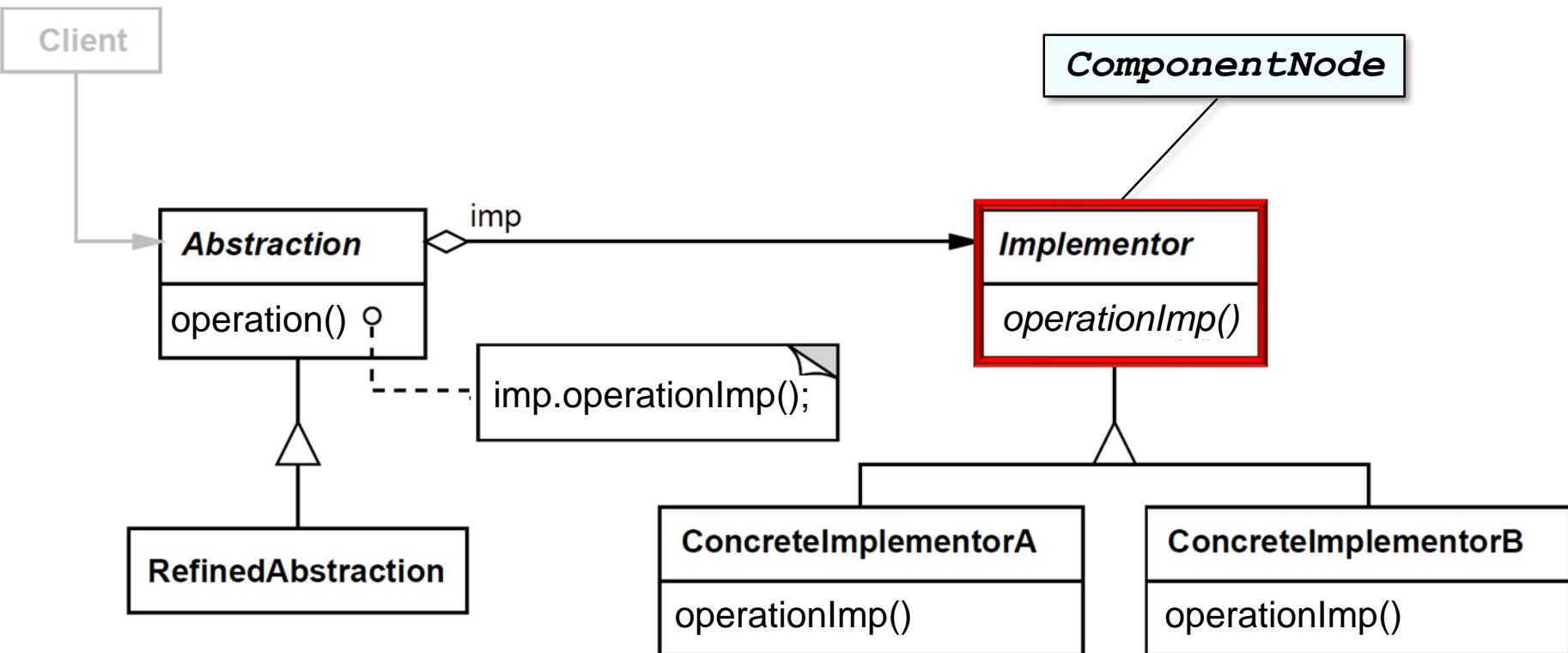
Structure & Participants



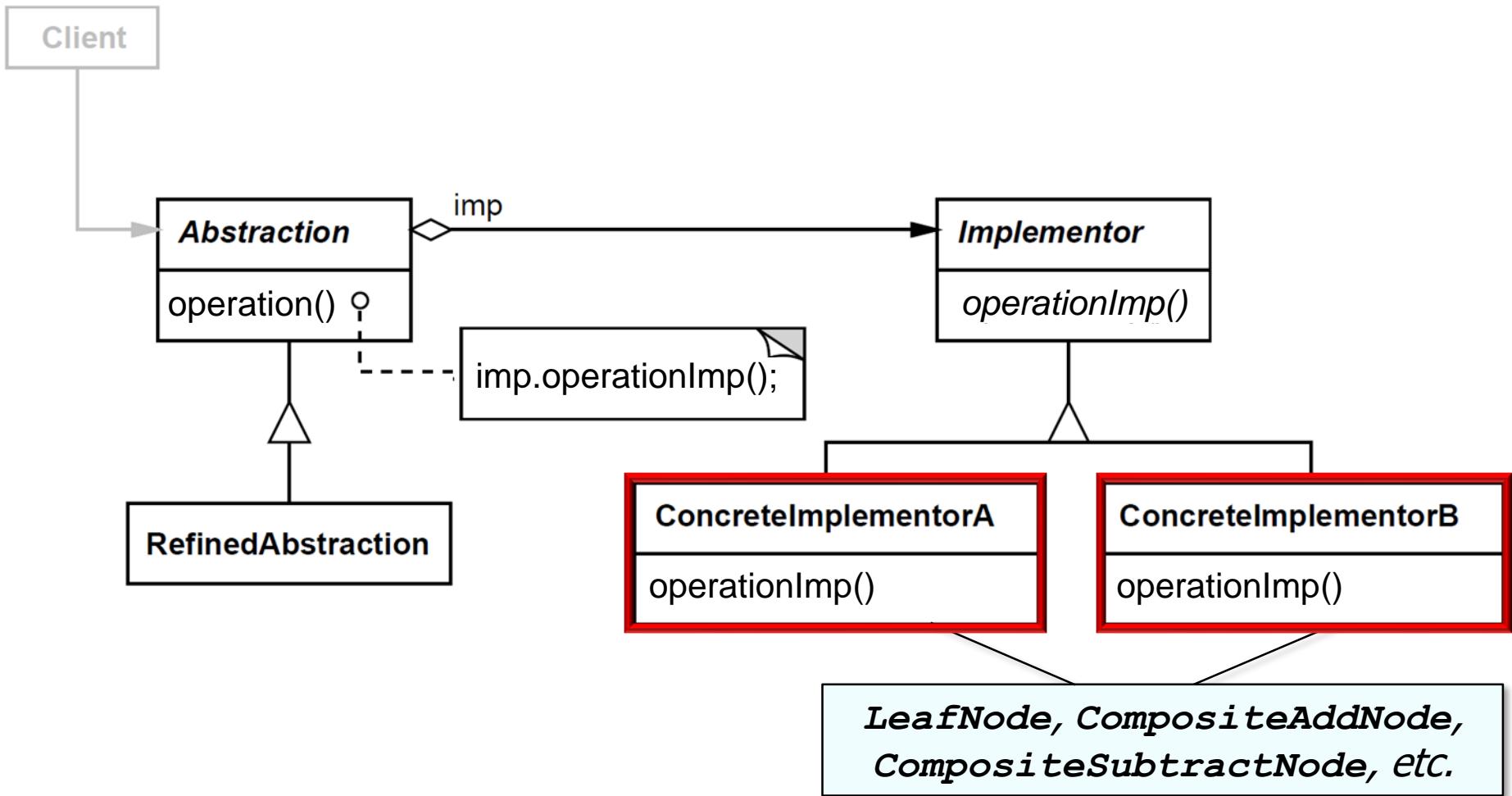
Structure & Participants



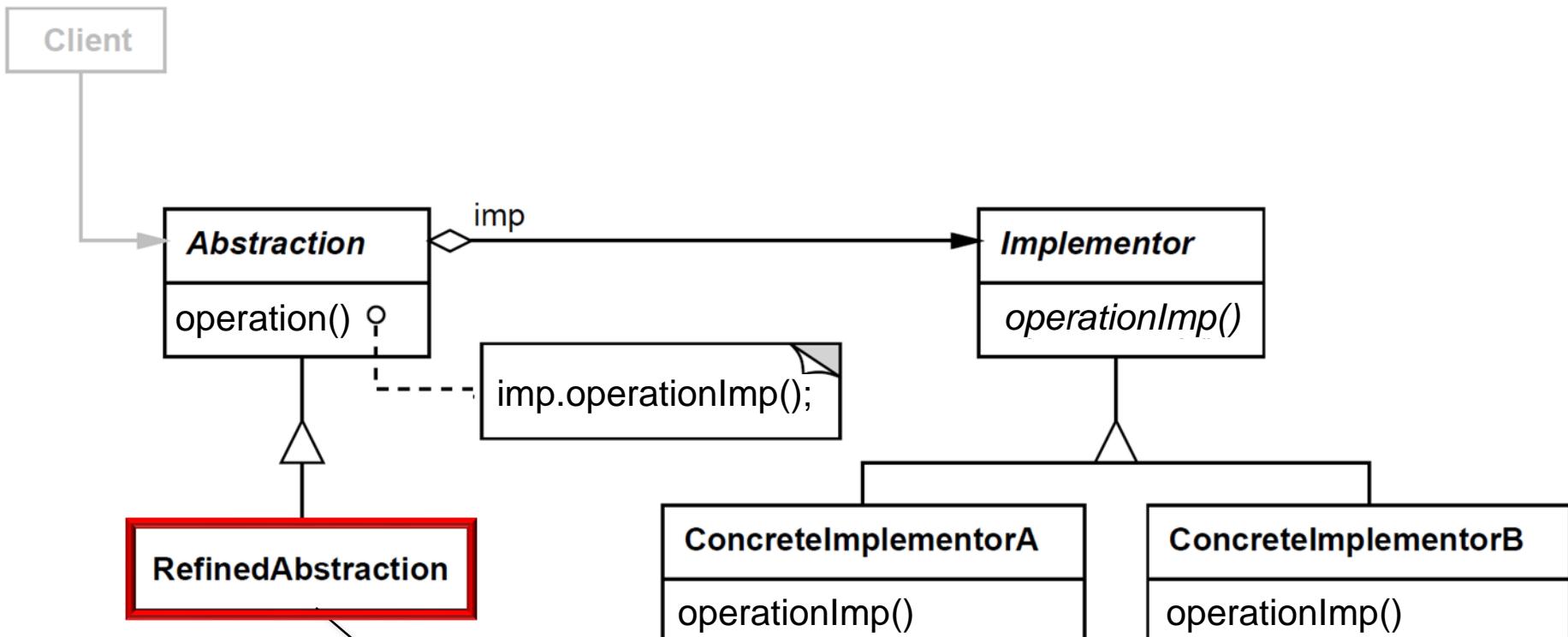
Structure & Participants



Structure & Participants



Structure & Participants



*SynchronizedExpressionTree,
InstrumentedExpressionTree, etc.*

Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy

```
class ExpressionTree
{
    private ComponentNode mRoot;

    public ExpressionTree(ComponentNode root) {
        mRoot = root;
        ...
    }

    public void accept(Visitor v) { mRoot.accept(v); }
}
```

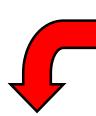
See [ExpressionTree/CommandLine/src/expressiontree/tree](#)

Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy

```
class ExpressionTree {  
    private ComponentNode mRoot;  
  
    public ExpressionTree(ComponentNode root) {  
        mRoot = root;  
  
        ...  
  
        public void accept(Visitor v) { mRoot.accept(v); }  
  
    }  
}
```

Stores the root of Composite implementor hierarchy



Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy

```
class ExpressionTree
{
    private ComponentNode mRoot;

    public ExpressionTree(ComponentNode root) {
        mRoot = root;
        ...
    }

    public void accept(Visitor v) { mRoot.accept(v); }
}
```

Pass in root of Composite
implementor hierarchy



Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy

```
class ExpressionTree
{
    private ComponentNode mRoot;

    public ExpressionTree(ComponentNode root) {
        mRoot = root;
        ...
    }

    public void accept(Visitor v) { mRoot.accept(v); }

}
```

Abstraction forwards to implementor via mRoot



Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy

```
class InstrumentedExpressionTree extends ExpressionTree {  
    public void accept(Visitor v) {  
        System.out.println("starting accept() call" + ...);  
        super.accept(v);  
        System.out.println("finished accept() call" + ...);  
    }  
    ...  
}
```

Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy

```
class InstrumentedExpressionTree extends ExpressionTree {  
    public void accept(Visitor v) {  
        System.out.println("starting accept() call" + ...);  
        super.accept(v);  
        System.out.println("finished accept() call" + ...);  
    }  
    ...  
}
```

Print logging messages both before & after call to accept()



Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy

```
class InstrumentedExpressionTree extends ExpressionTree {  
    public void accept(Visitor v) {  
        System.out.println("starting accept() call" + ...);  
        super.accept(v);  
        System.out.println("finished accept() call" + ...);  
    }  
    ...  
  
class SynchronizedExpressionTree extends ExpressionTree {  
    public void accept(Visitor v) {  
        synchronized(this) {  
            super.accept(v);  
        }  
    }  
    ...
```

Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy

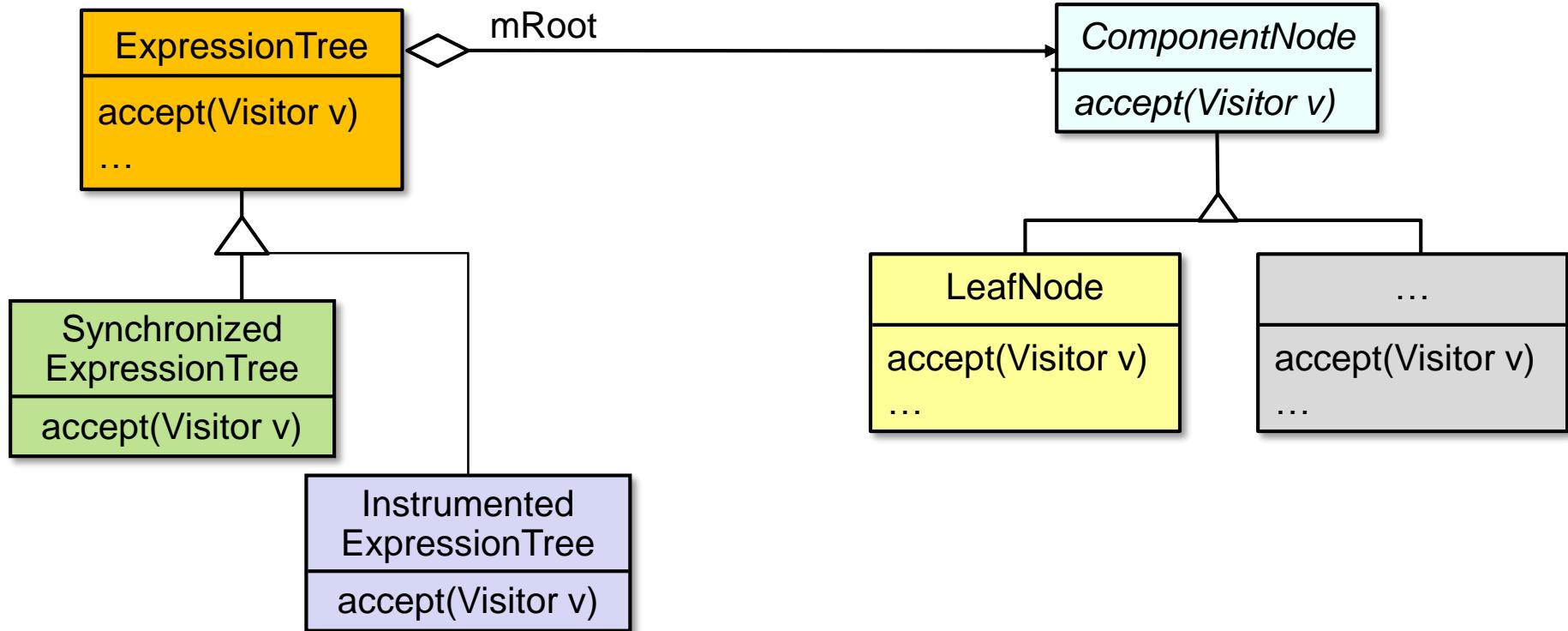
```
class InstrumentedExpressionTree extends ExpressionTree {  
    public void accept(Visitor v) {  
        System.out.println("starting accept() call" + ...);  
        super.accept(v);  
        System.out.println("finished accept() call" + ...);  
    }  
    ...  
  
class SynchronizedExpressionTree extends ExpressionTree {  
    public void accept(Visitor v) {  
        synchronized(this) {  
            super.accept(v);  
        }  
    }  
    ...
```



Synchronize the call to accept()

Bridge example in Java

- Separate expression tree abstraction from composite implementor hierarchy



These changes in service behavior don't affect the implementor hierarchy

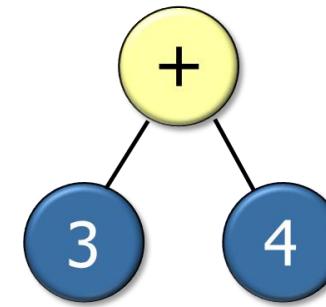
Bridge example in Java

- Encapsulate sources of variability in expression tree construction & use

```
ExpressionTree exprTree  
  (new CompositeAddNode  
    (new LeafNode (3),  
     new LeafNode (4)));
```



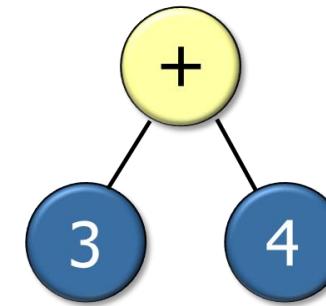
Hide use of complex recursive
Composite internal structure
behind a stable *Bridge* API



Bridge example in Java

- Encapsulate sources of variability in expression tree construction & use

```
ExpressionTree exprTree  
  (new CompositeAddNode  
    (new LeafNode (3),  
     new LeafNode (4)));
```



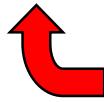
Replace *Composite* implementation
with *TreeNode* implementation

```
ExpressionTree exprTree  
  (new TreeNode  
    ('+',  
     new TreeNode(3),  
     new TreeNode(4)));
```

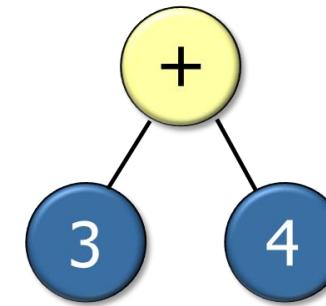
Bridge example in Java

- Encapsulate sources of variability in expression tree construction & use

```
ExpressionTree exprTree  
  (makeExpressionTree ("3+4")) ;
```



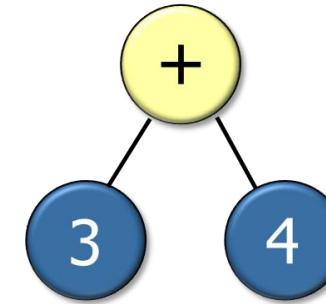
We can apply a **creational pattern**
to reduce client dependencies on
implementation variability



Bridge example in Java

- Encapsulate sources of variability in expression tree construction & use

Regardless of which implementation or abstraction was used, however, we can iterate through all tree elements without concern for how tree is structured internally

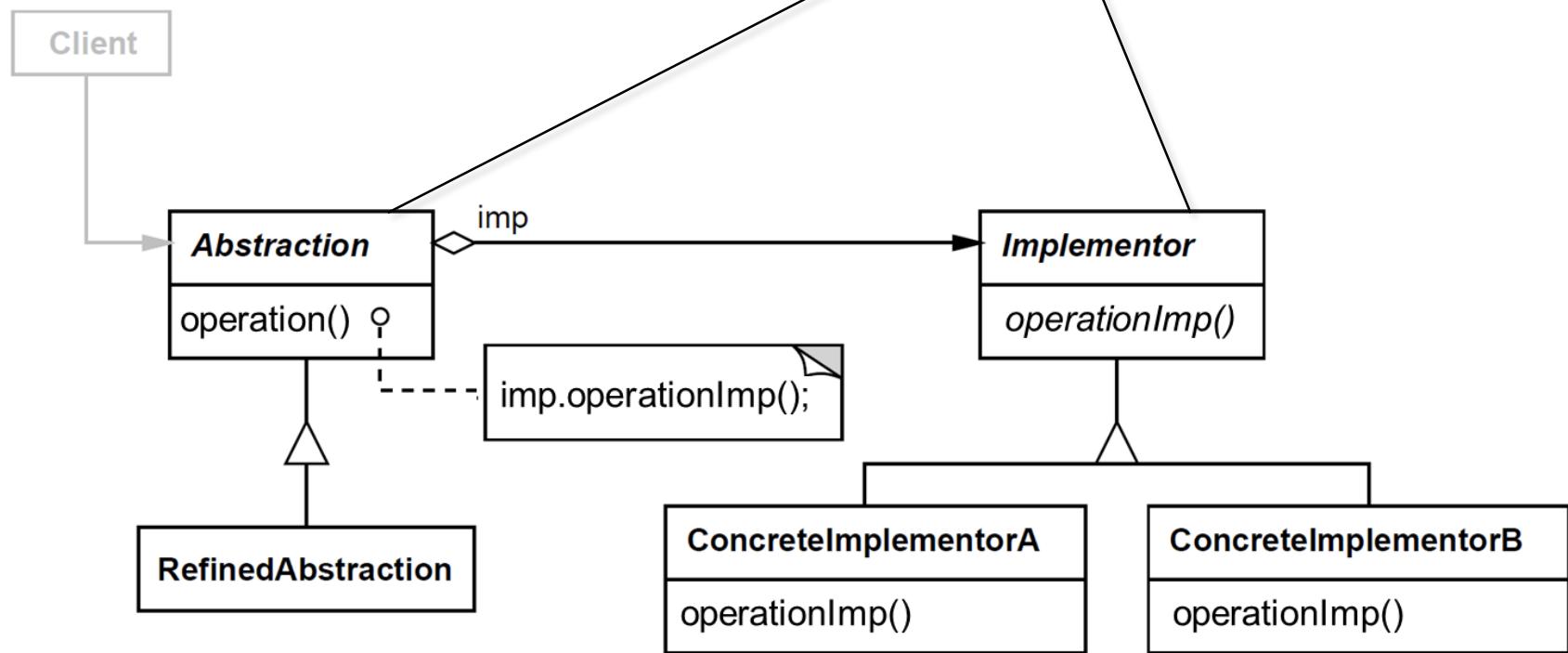


for (Iterator<ExpressionTree> it =
 exprTree.iterator();
 it.hasNext();)
 doSomethingWithEachNode(it.next());

Consequences

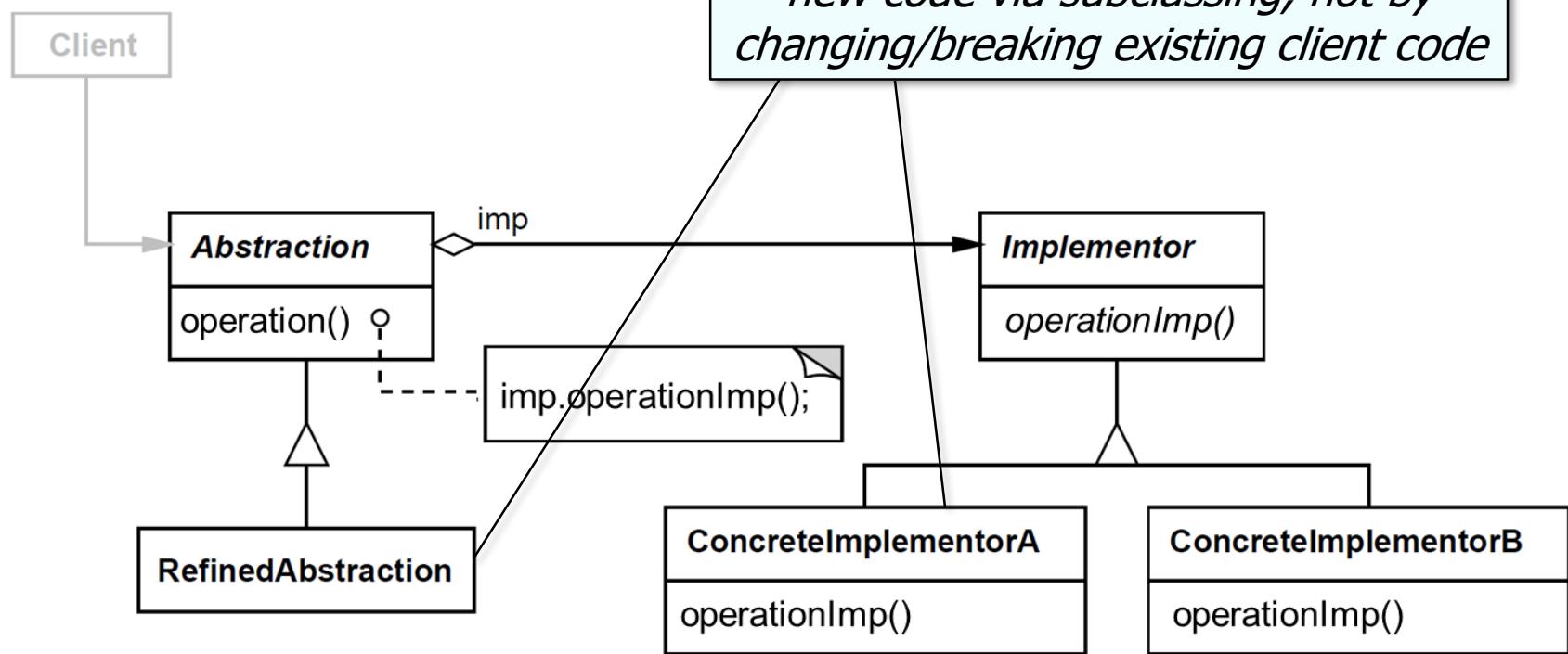
- + Abstraction & implementor hierarchy are decoupled
 - Can evolve separately by applying *Open/Closed Principle*

Enable software to be open for extension (via implementor hierarchy), but closed for modification (via stable abstraction API)



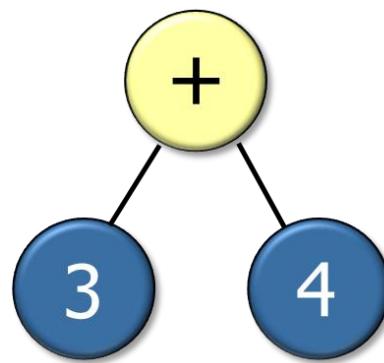
Consequences

- + Abstraction & implementor hierarchy are decoupled
- Can evolve separately by applying *Open/Closed Principle*



Consequences

- + Implementors can vary at design-time or runtime



```
ExpressionTree exprTree  
(new CompositeAddNode  
    (new LeafNode (3),  
     new LeafNode (4)));
```

versus

```
ExpressionTree exprTree  
(new TreeNode  
    ('+',  
     new TreeNode(3),  
     new TreeNode(4)));
```

Consequences

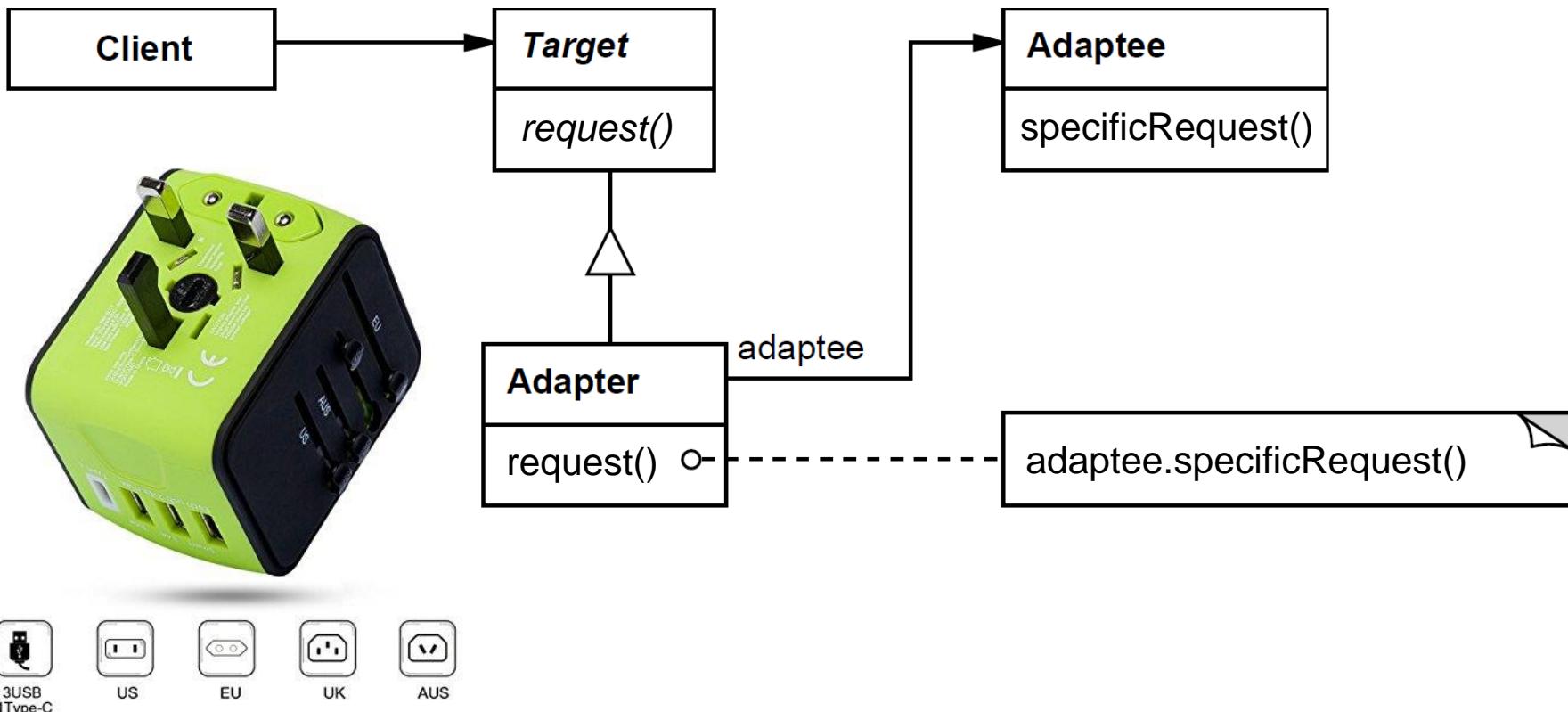
- “One-size-fits-all” abstraction & implementor interfaces



See en.wikipedia.org/wiki/Procrustes#Cultural_references

Consequences

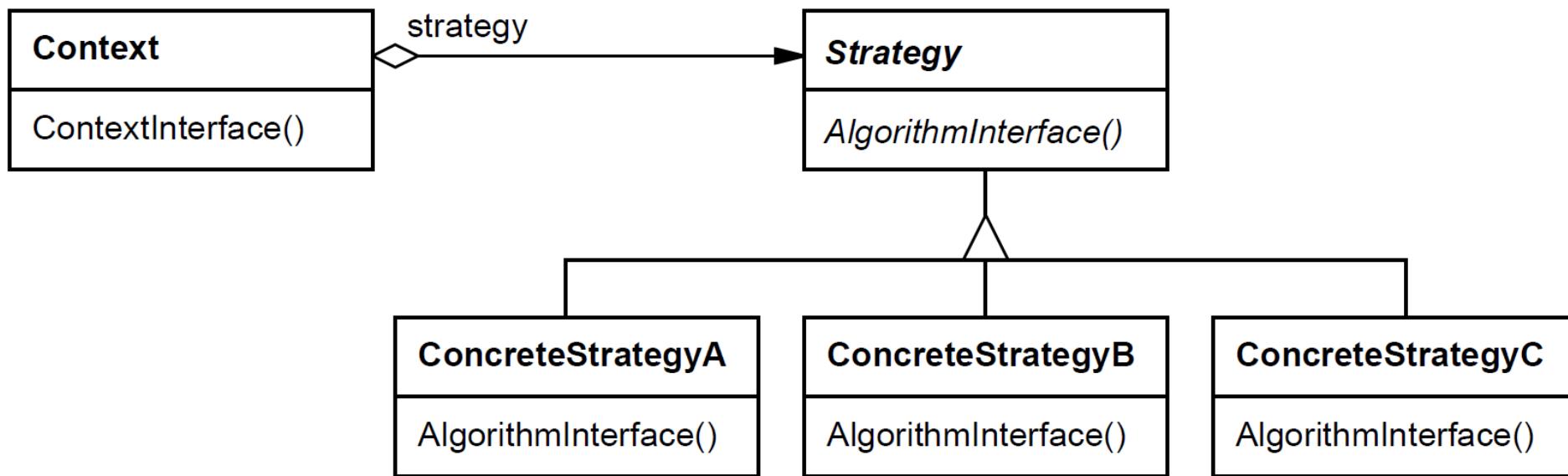
- “One-size-fits-all” abstraction & implementor interfaces
 - Can be alleviated via other patterns, e.g.
 - *Adapter* – Makes existing classes work w/others without modifying code



en.wikipedia.org/wiki/Adapter_pattern has more on *Adapter*

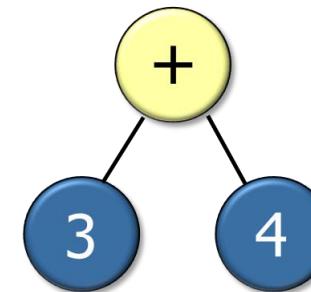
Consequences

- “One-size-fits-all” abstraction & implementor interfaces
 - Can be alleviated via other patterns, e.g.
 - *Adapter* – Makes existing classes work w/others without modifying code
 - *Strategy* – Lets the algorithm vary independently from clients that use it

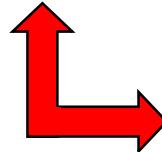


Implementation considerations

- Creating the right abstraction or implementor
 - Often addressed by using Creational patterns
 - e.g., *Factory Method* or *Builder*



```
class Multiply extends Operator {  
    ...  
    ComponentNode build() {  
        return new CompositeMultiplyNode(left.build(), right.build());  
    }  
}
```

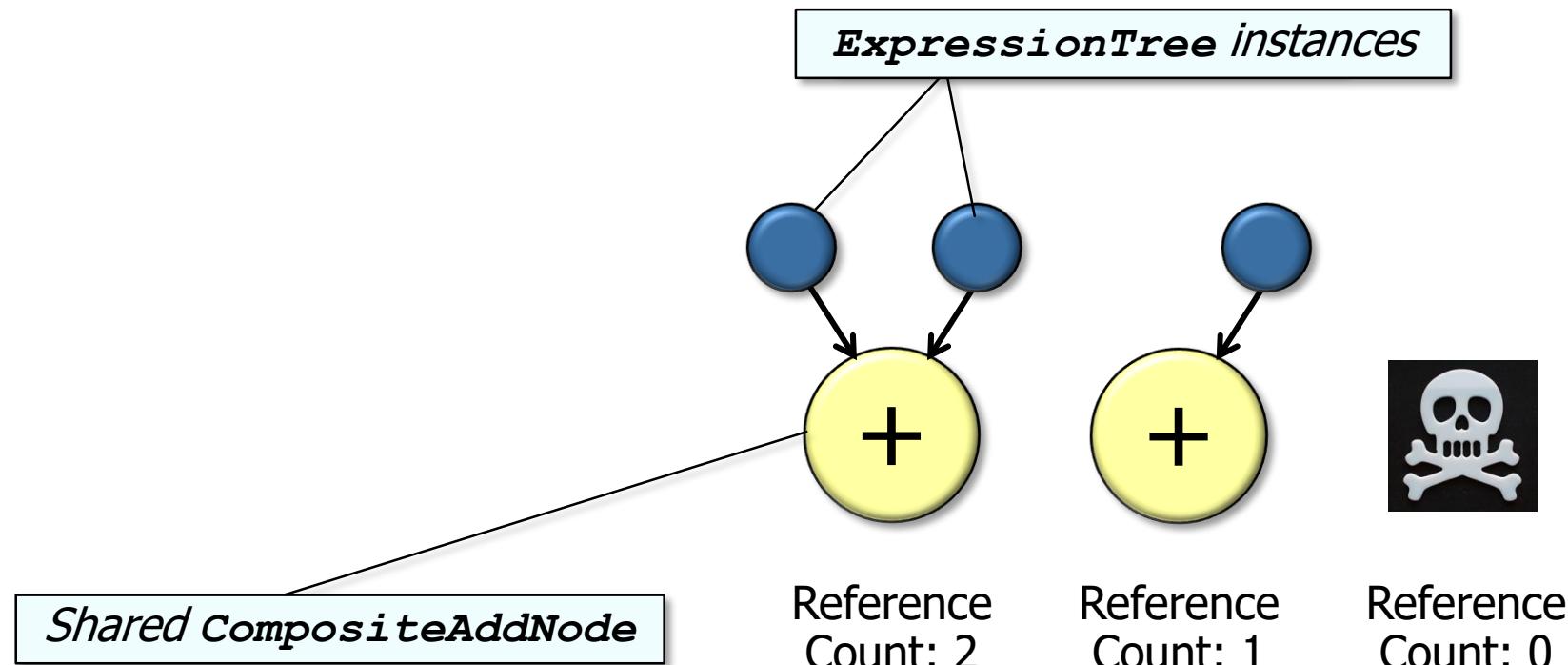


Build corresponding component nodes

```
class Number extends Expr {  
    ...  
    ComponentNode build()  
    { return new LeafNode(item); }  
}
```

Implementation considerations

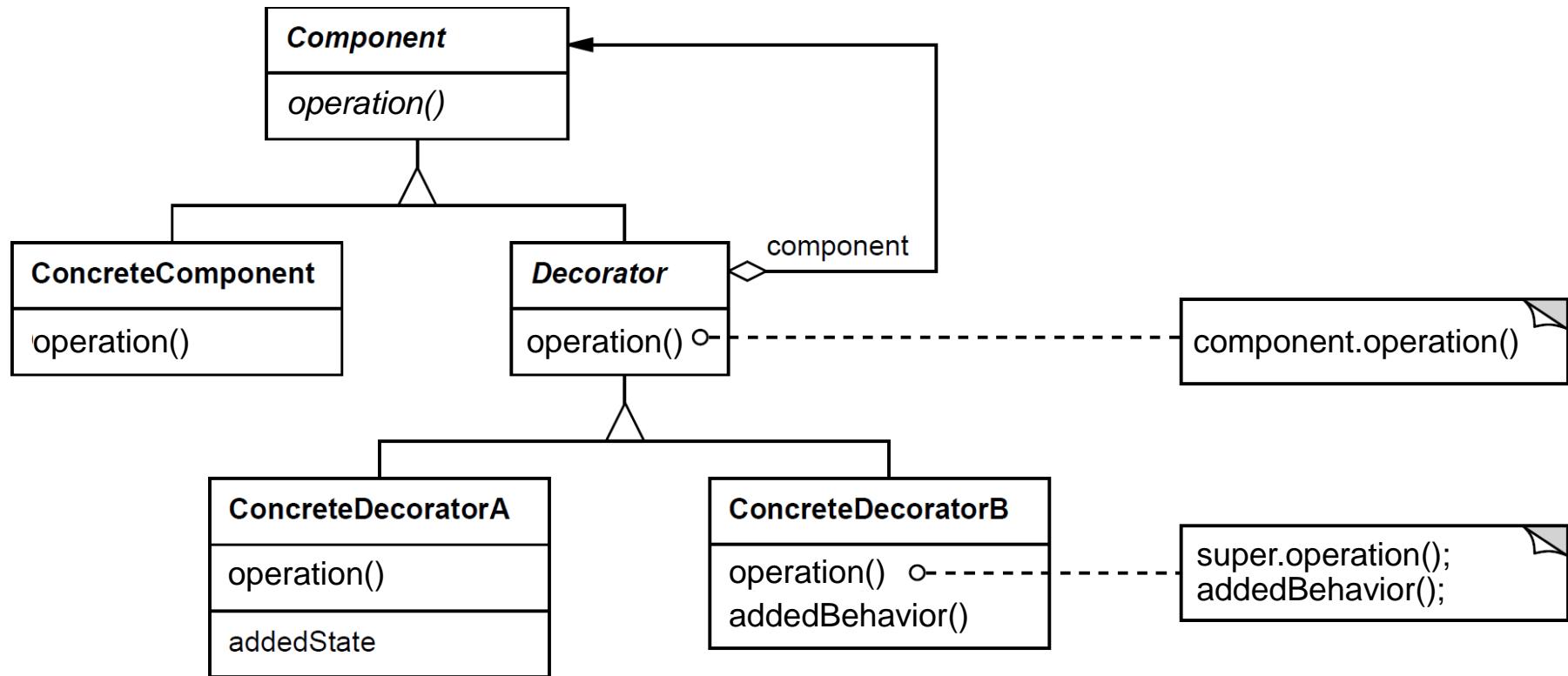
- Sharing implementors & reference counting
 - e.g., C++11/Boost `shared_ptr`



See en.wikipedia.org/wiki/Smart_pointer#shared_ptr_and_weak_ptr

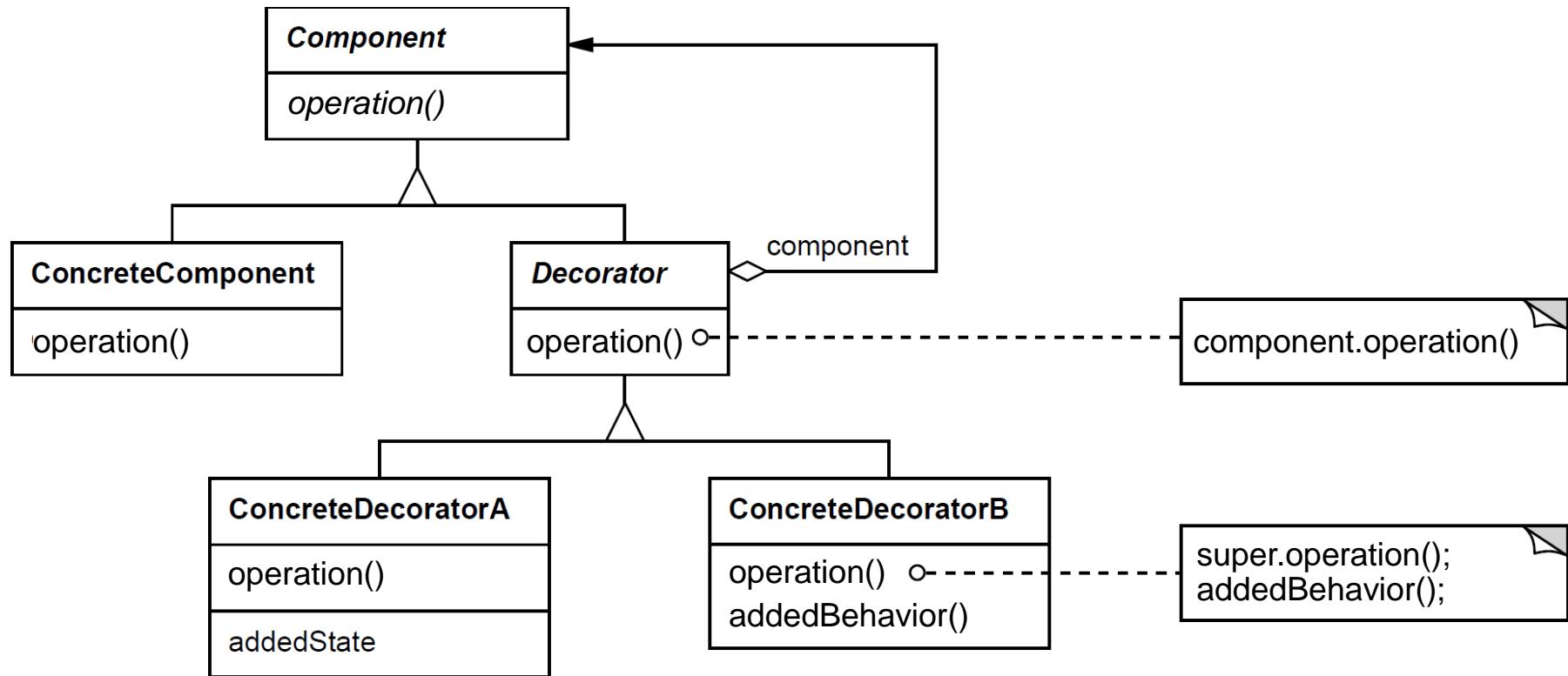
Implementation considerations

- More dynamic uses of *Bridge* may be better implemented as a *Decorator*



Implementation considerations

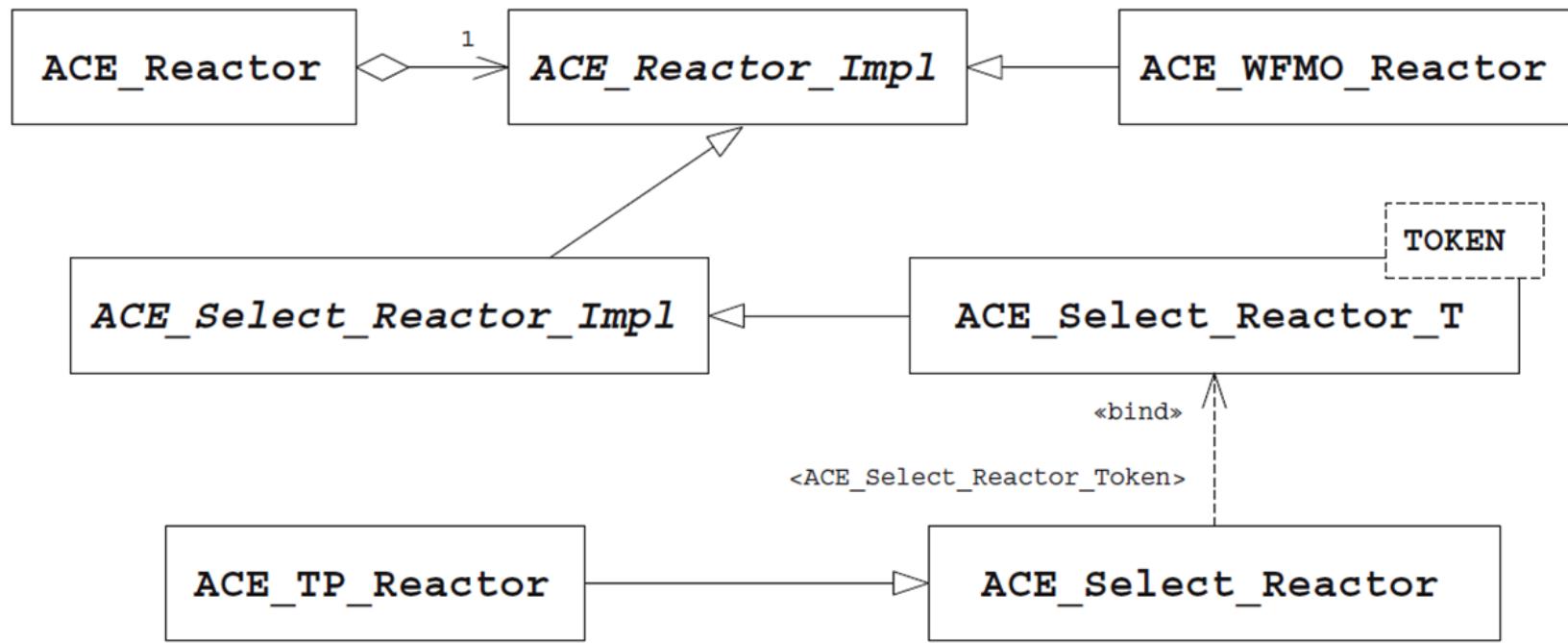
- More dynamic uses of *Bridge* may be better implemented as a *Decorator*



- Decorator* enables client-specified embellishment of a core object by recursively wrapping it (possibly more than once) dynamically at runtime

Known Uses

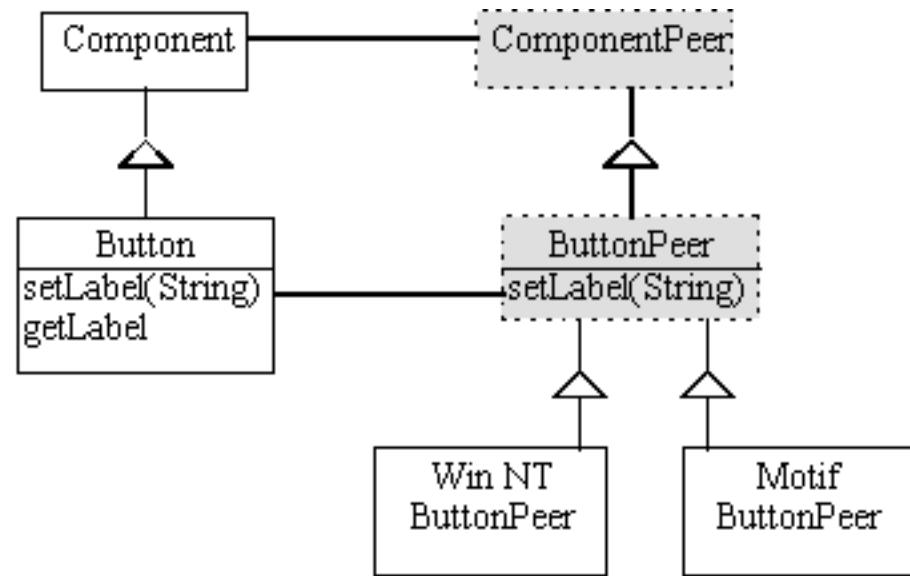
- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework



Bridge is used more in C++ than in Java (which uses interfaces & factories)

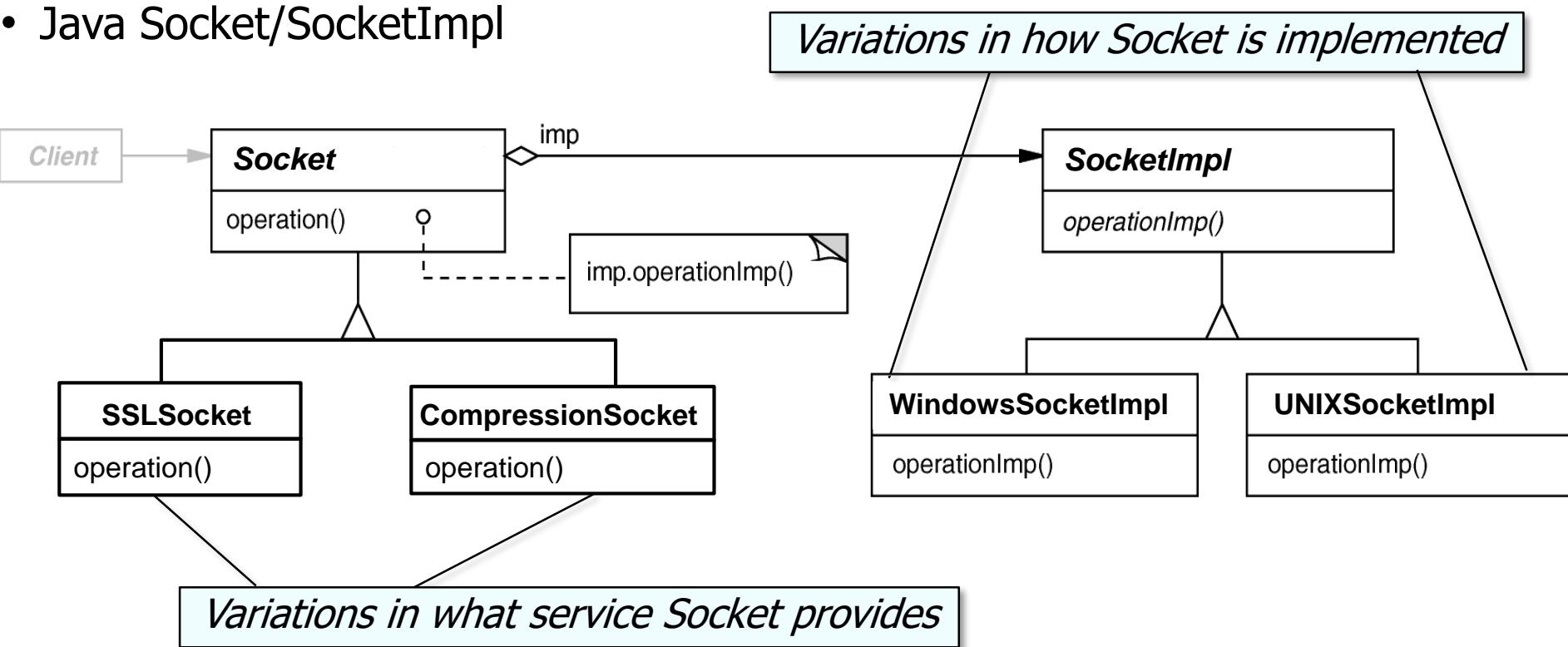
Known Uses

- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework
- AWT Component/ComponentPeer



Known Uses

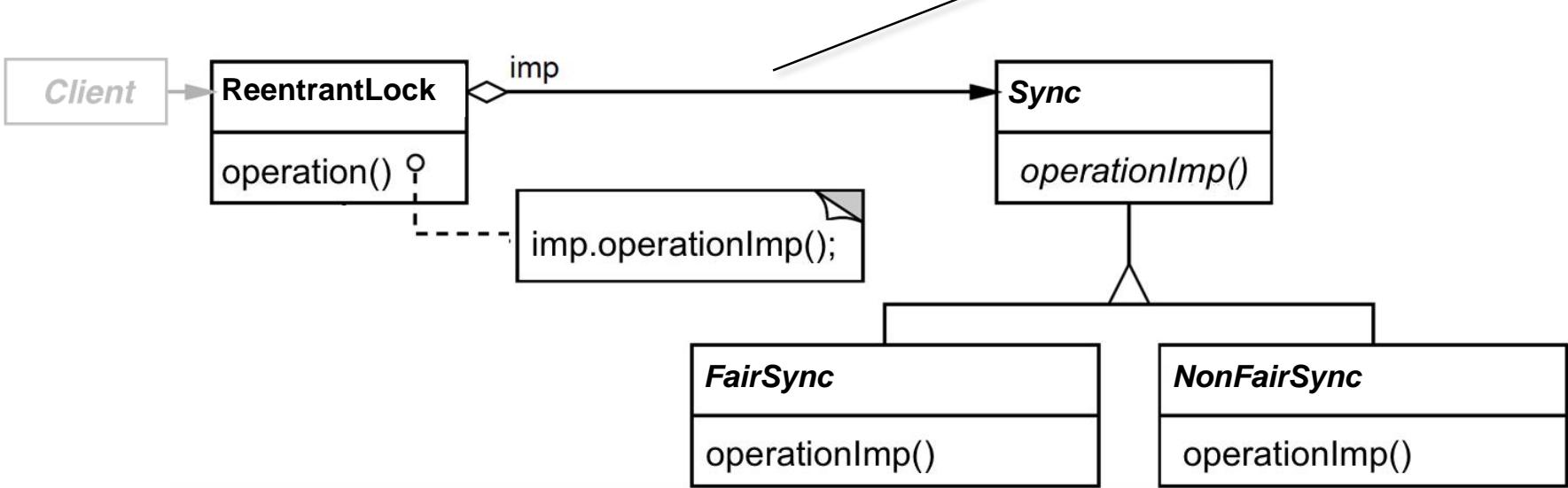
- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework
- AWT Component/ComponentPeer
- Java Socket/SocketImpl



Known Uses

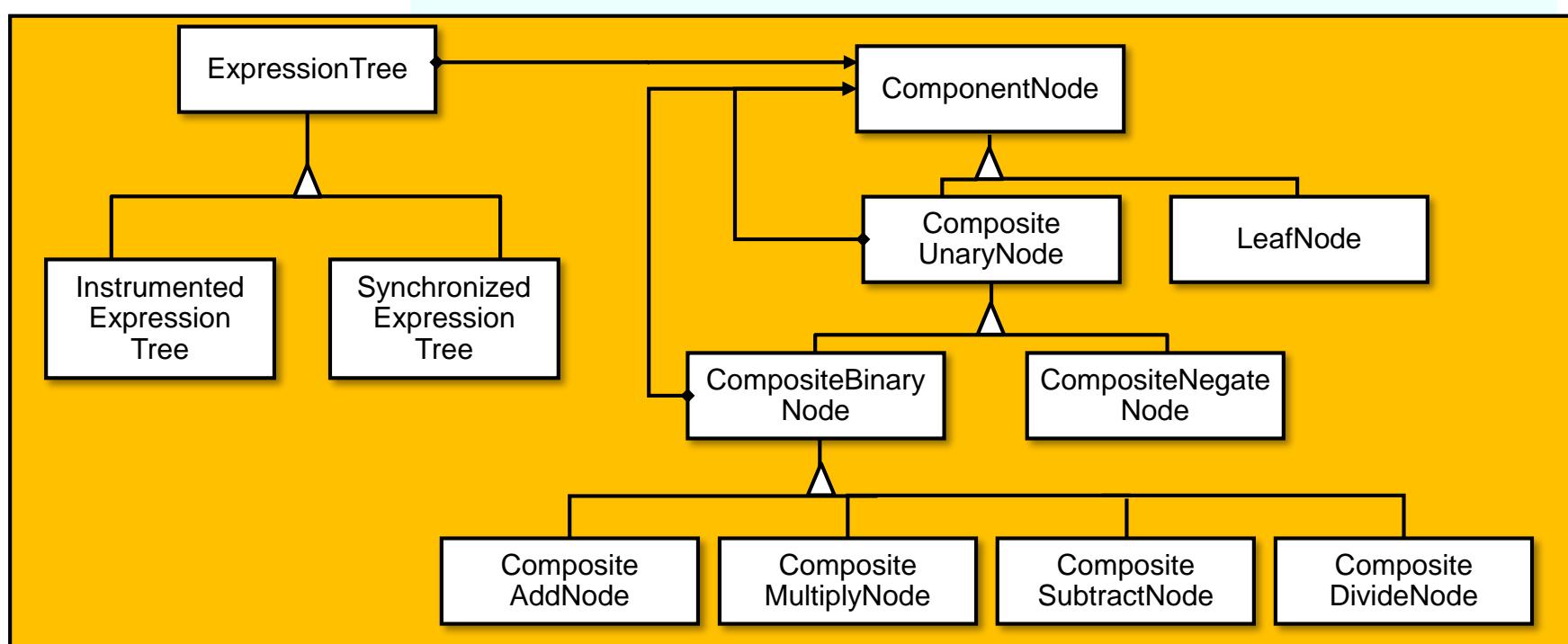
- ET++ Window/WindowPort
- libg++ Set/{LinkedList, HashTable}
- ACE Reactor framework
- AWT Component/ComponentPeer
- Java Socket/SocketImpl
- Java synchronizers

Decouples synchronizer interface from its implementation so fair & non-fair semantics can be supported uniformly



Summary of the Bridge Pattern

Bridge decouples the expression tree programming API from its behavior & implementation & enables transparent extensibility



Bridge

Composite

Bridge Composite is an example of a "pattern compound"

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

End of the
Bridge Pattern

The Interpreter Pattern

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software
Integrated Systems

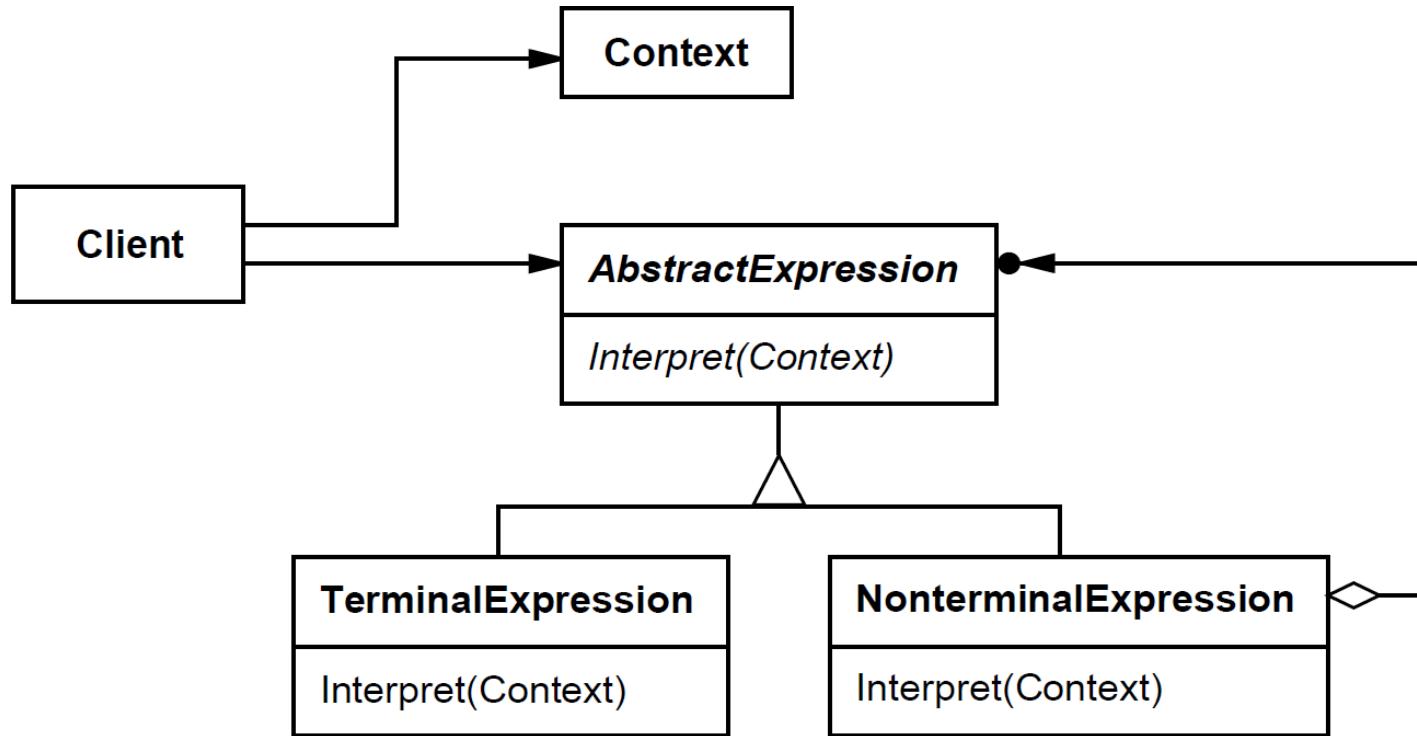
Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives

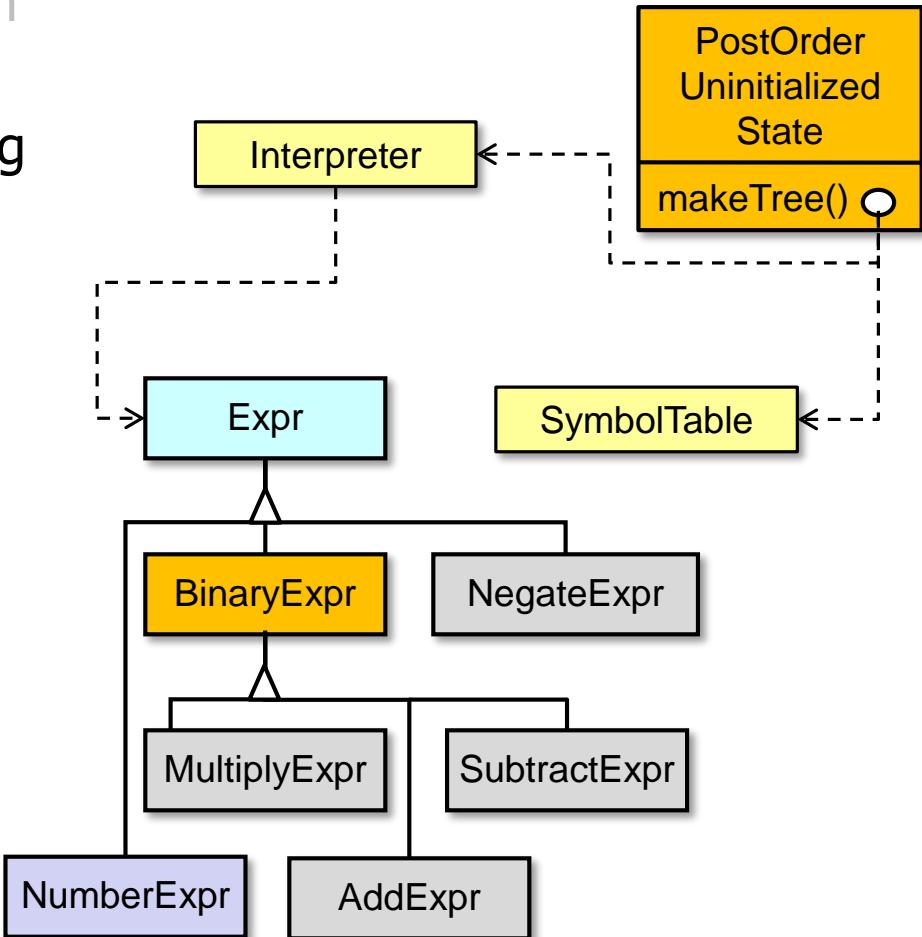
Learning Objectives

- Understand the *Interpreter* pattern



Learning Objectives

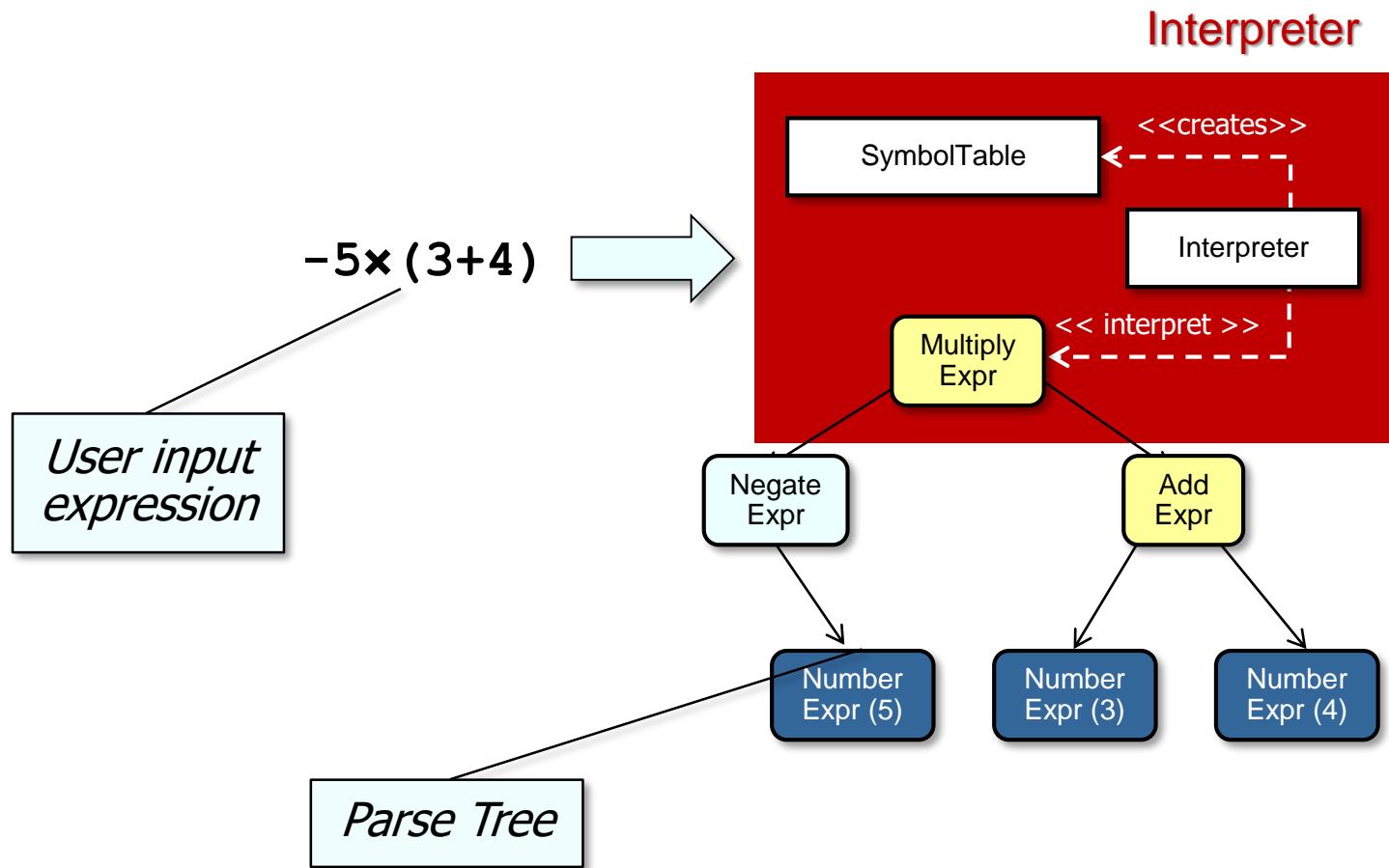
- Understand the *Interpreter* pattern
- Recognize how *Interpreter* can be applied to automate the processing of input expressions from users



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

A Pattern for Flexibly Processing User Input

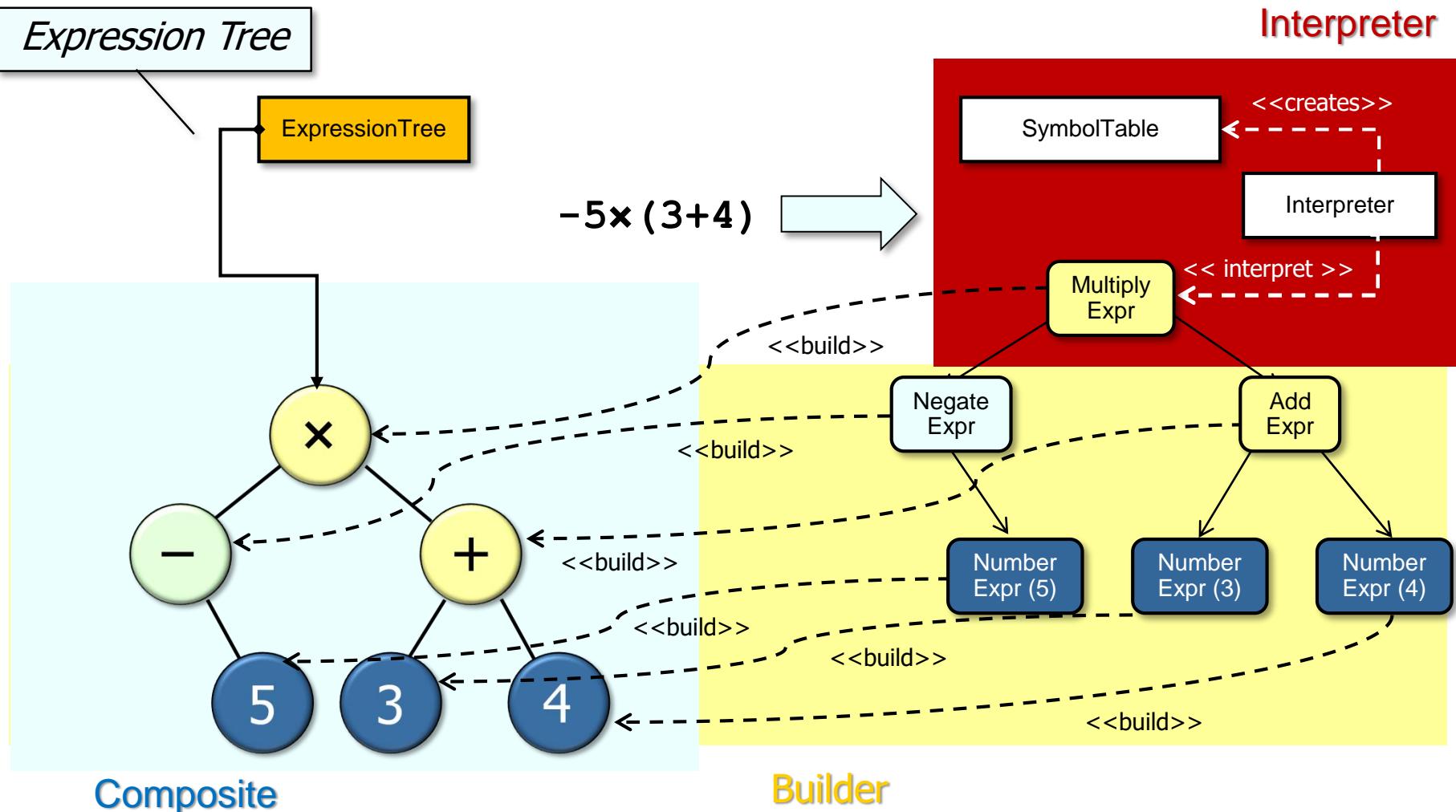
Purpose: Process a user input expression & build a parse tree



Interpreter automates the parsing of input expressions from users

A Pattern for Flexibly Processing User Input

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

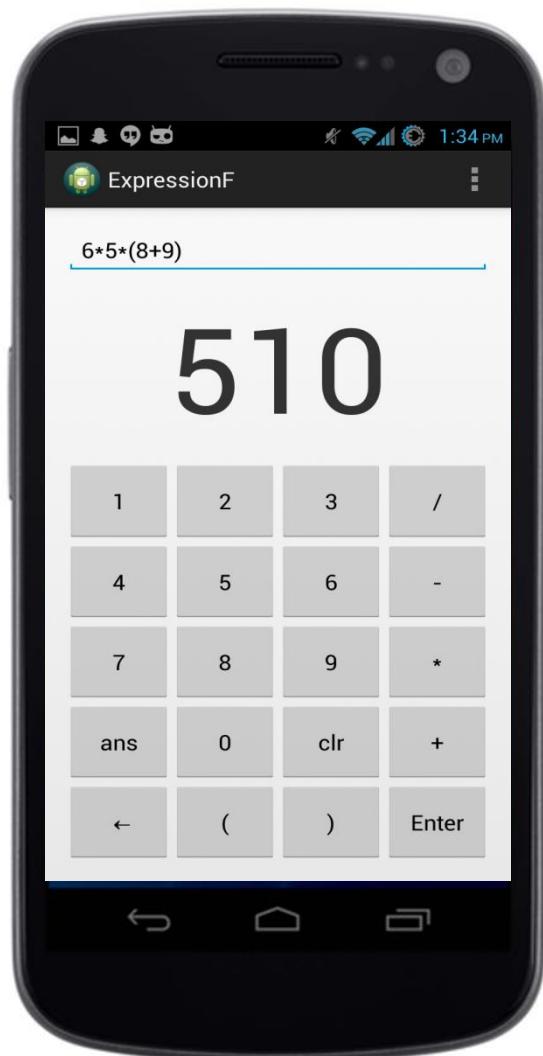
```
Console X
Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Jul 15, 2013 12:51:24 PM)

1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order| level-order]
0. quit

> format in-order

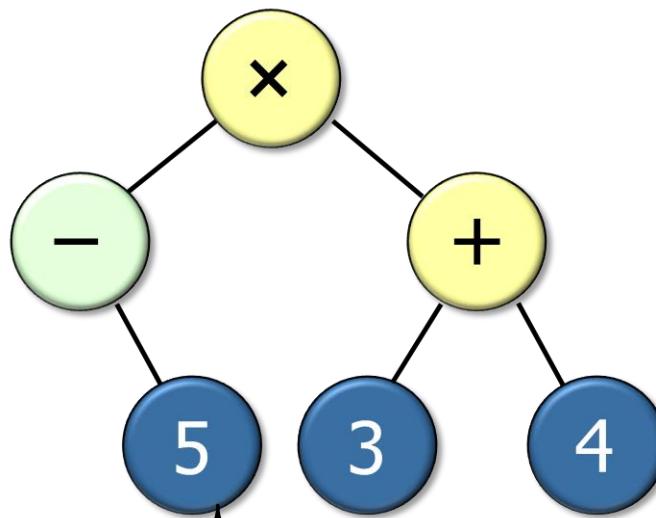
1. expr [expression]
2a. eval [post-order]
2b. print [in-order | pre-order | post-order| level-order]
0b. set [post-order]
0c. quit

> expr 6*5*(8+9)
```



Context: OO Expression Tree Processing App

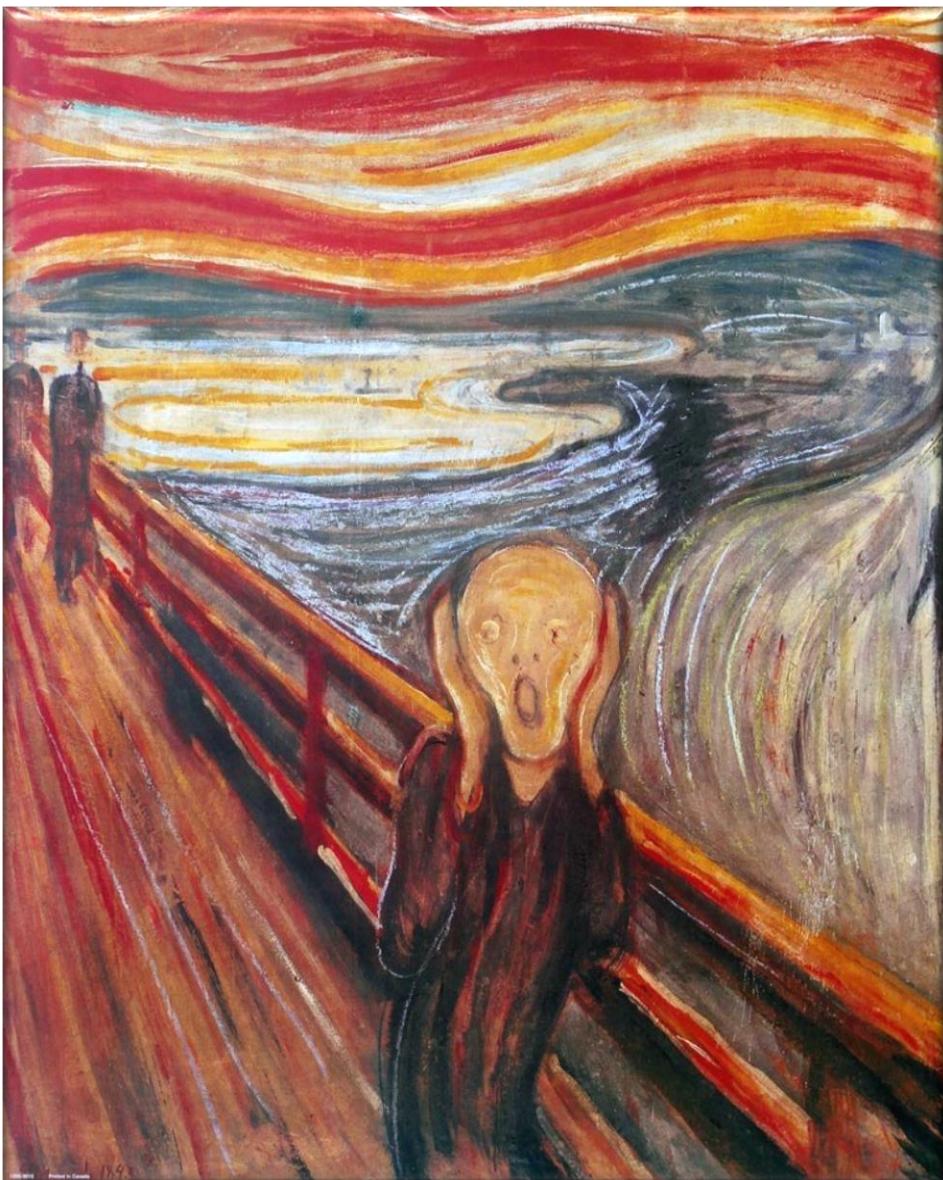
- The expression tree processing app also receives input in a variety of formats
 - e.g., pre-order, in-order, level-order, post-order, etc.



“pre-order” input expression = $x - 5 + 34$
“post-order” input expression = $5 ~ 34 + x$
“level-order” input expression = $x - + 5 3 4$
“in-order” input expression = $- 5 x (3 + 4)$

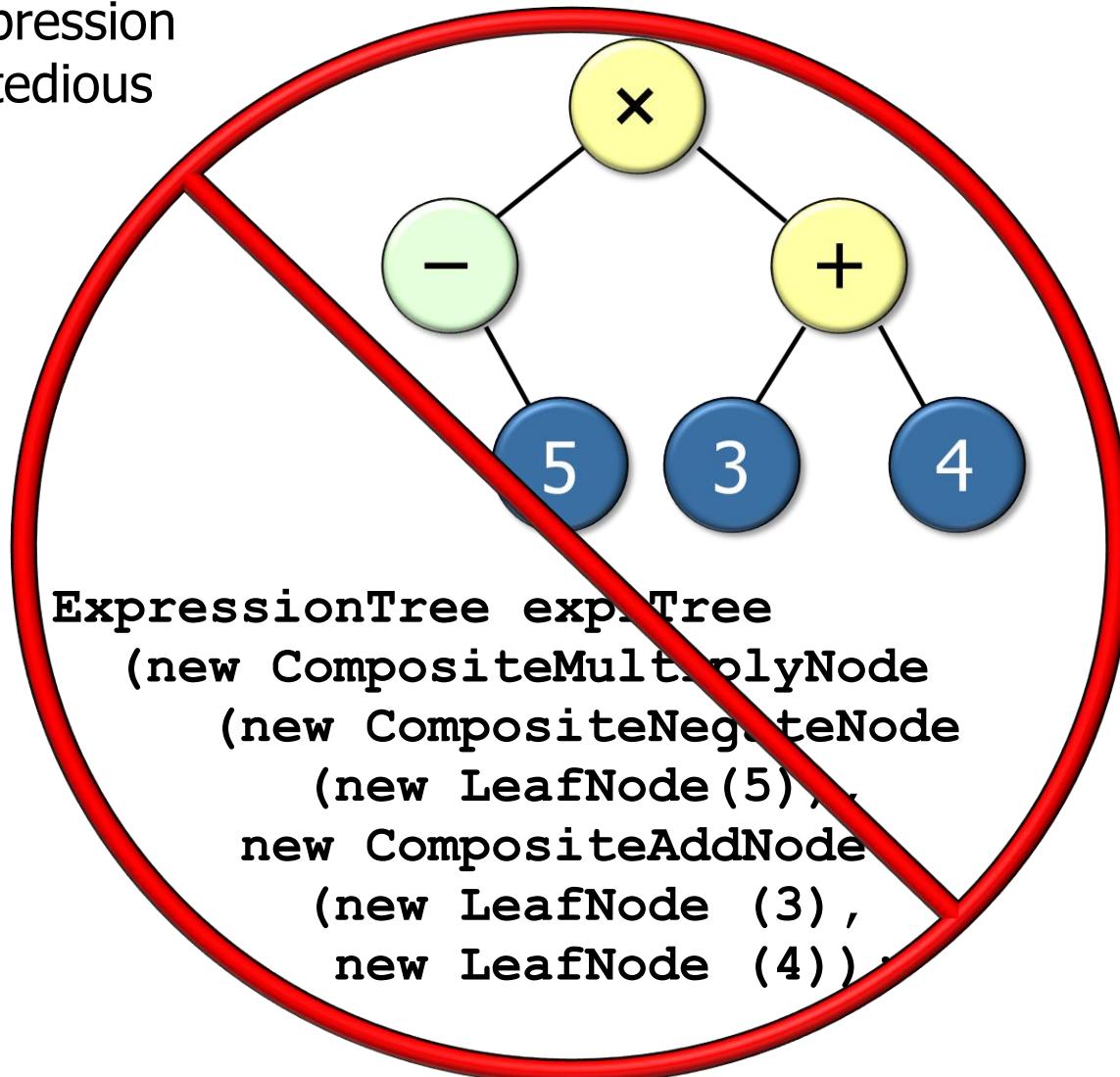
Problem: Inflexible Expression Input Processing

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



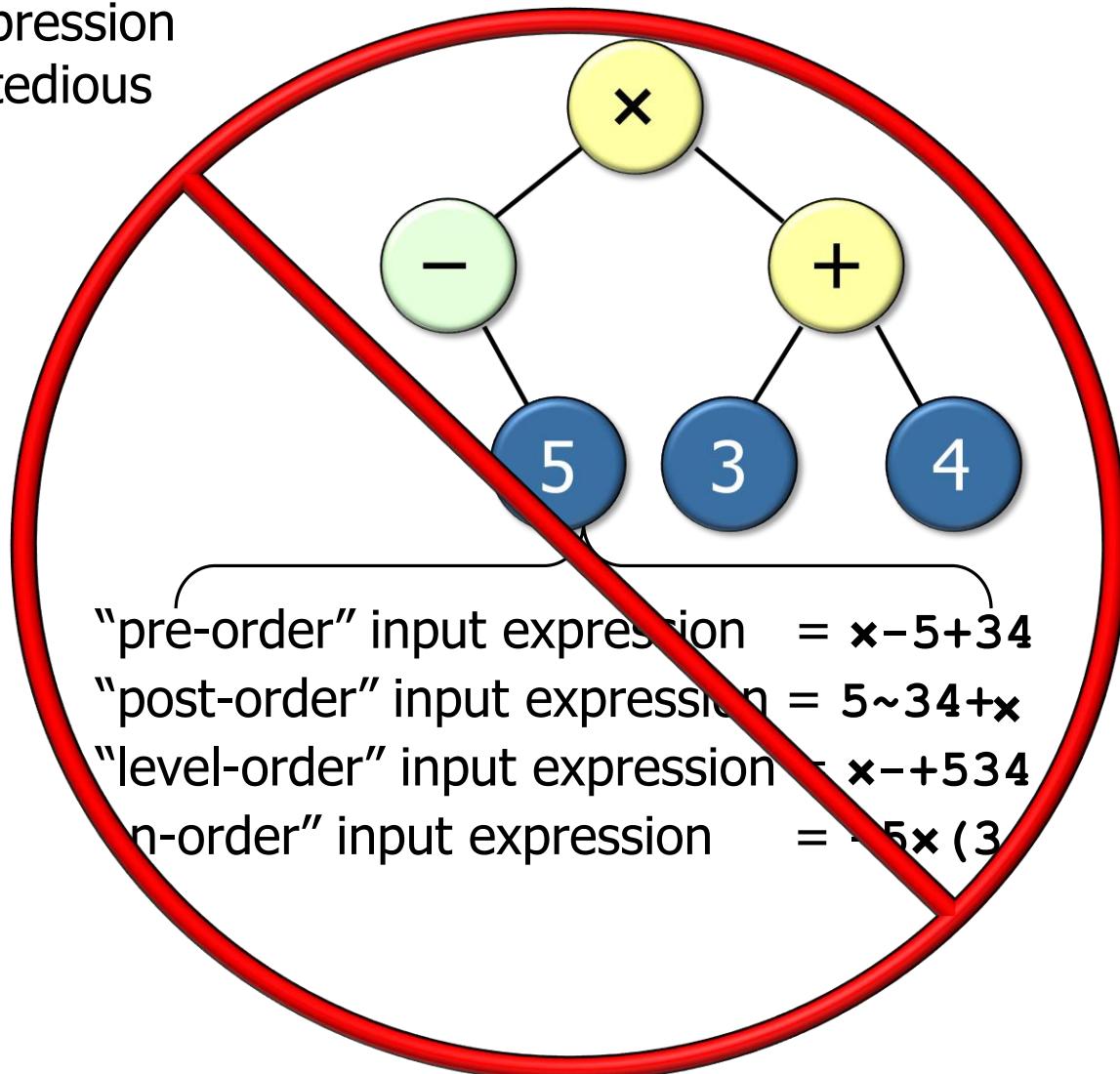
Problem: Inflexible Expression Input Processing

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



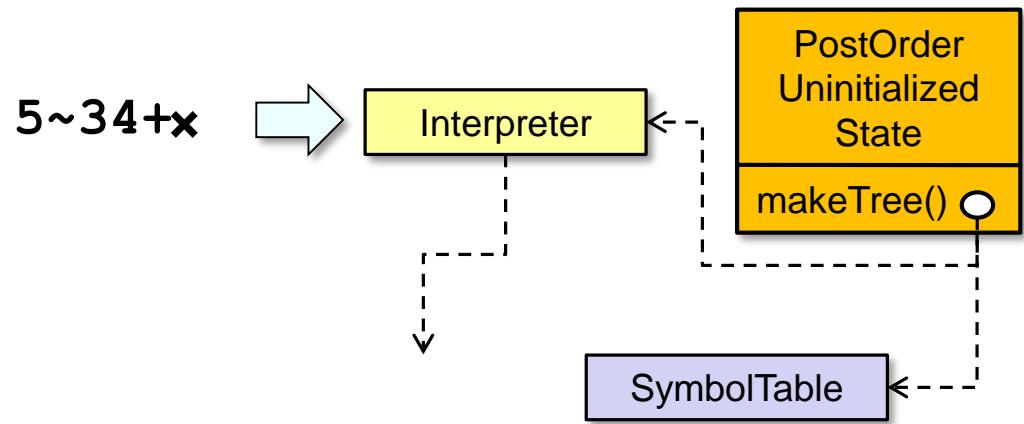
Problem: Inflexible Expression Input Processing

- Requiring users to create expression trees manually is extremely tedious & 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



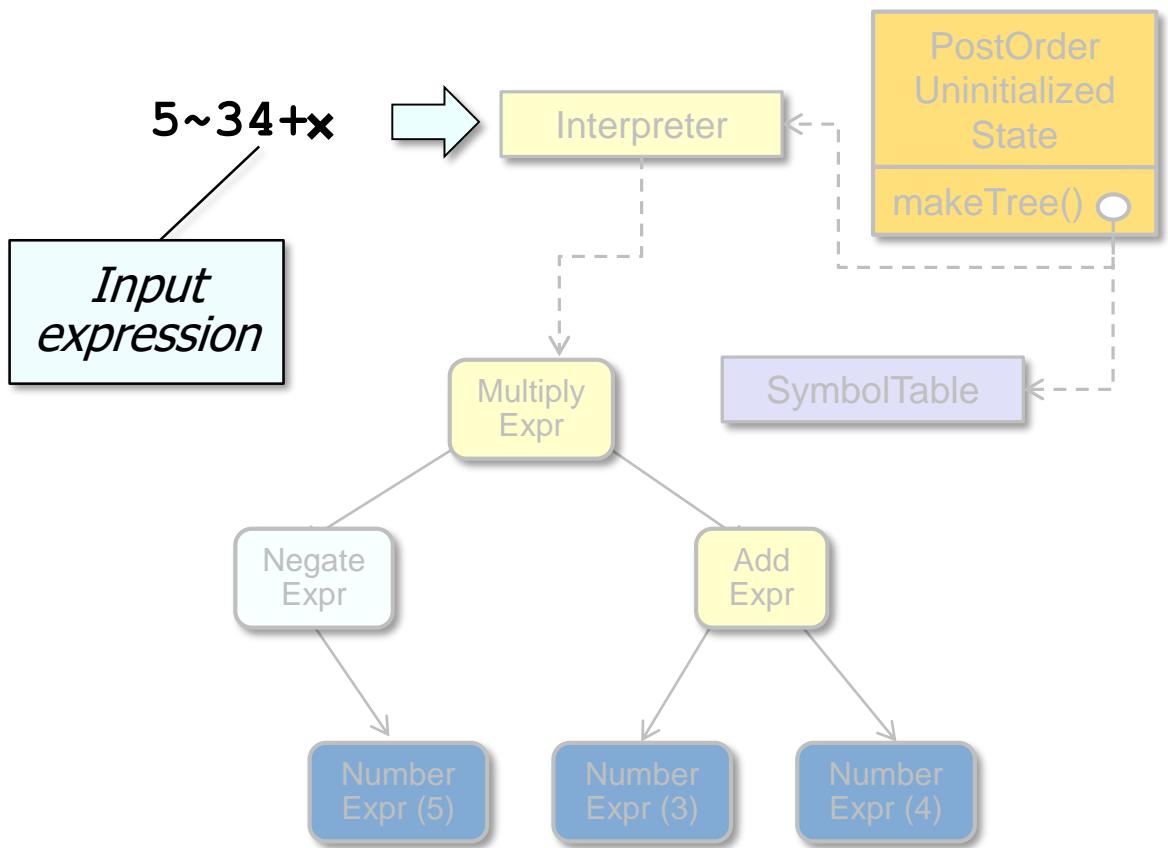
Solution: Create Interpreter to Process Input

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



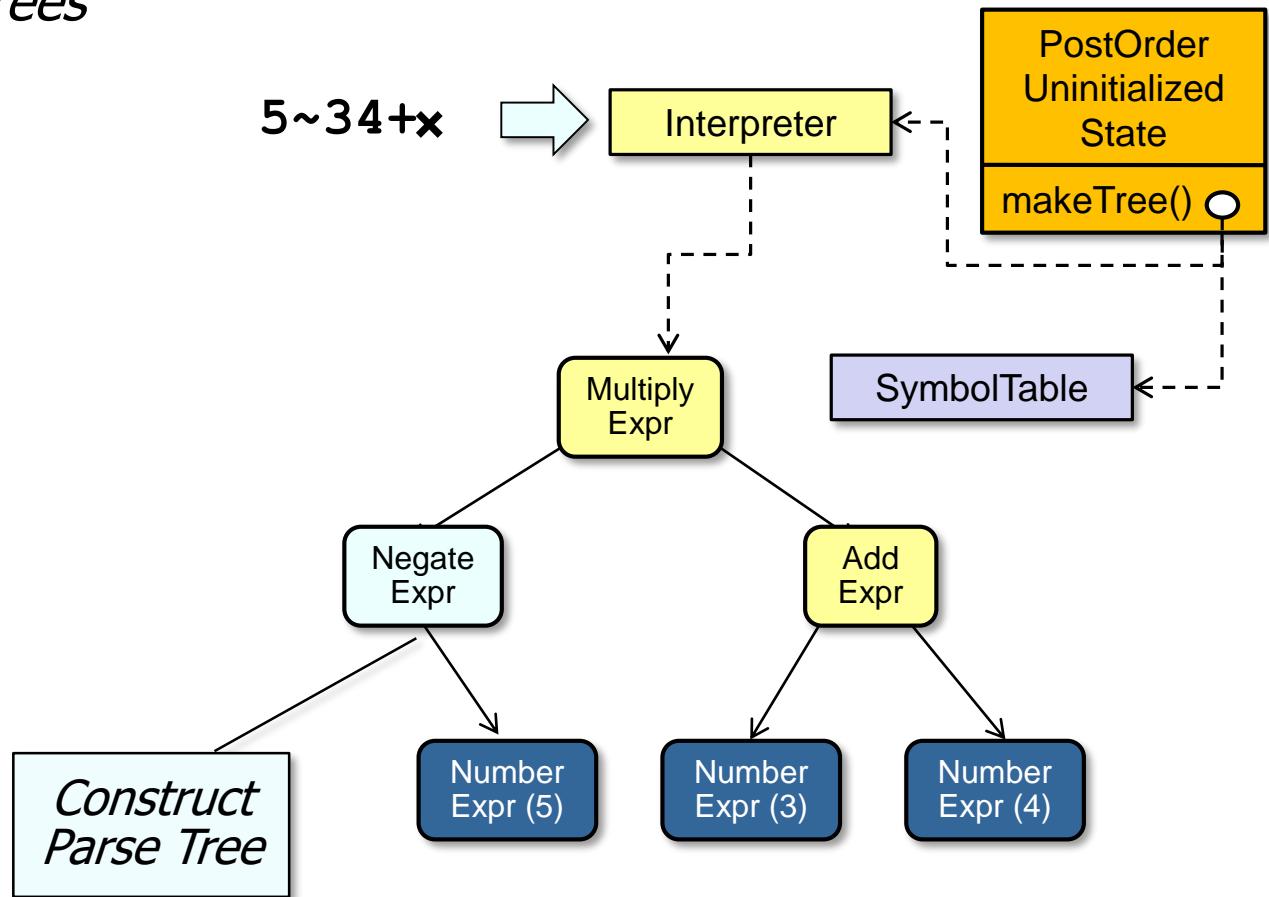
Solution: Create Interpreter to Process Input

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



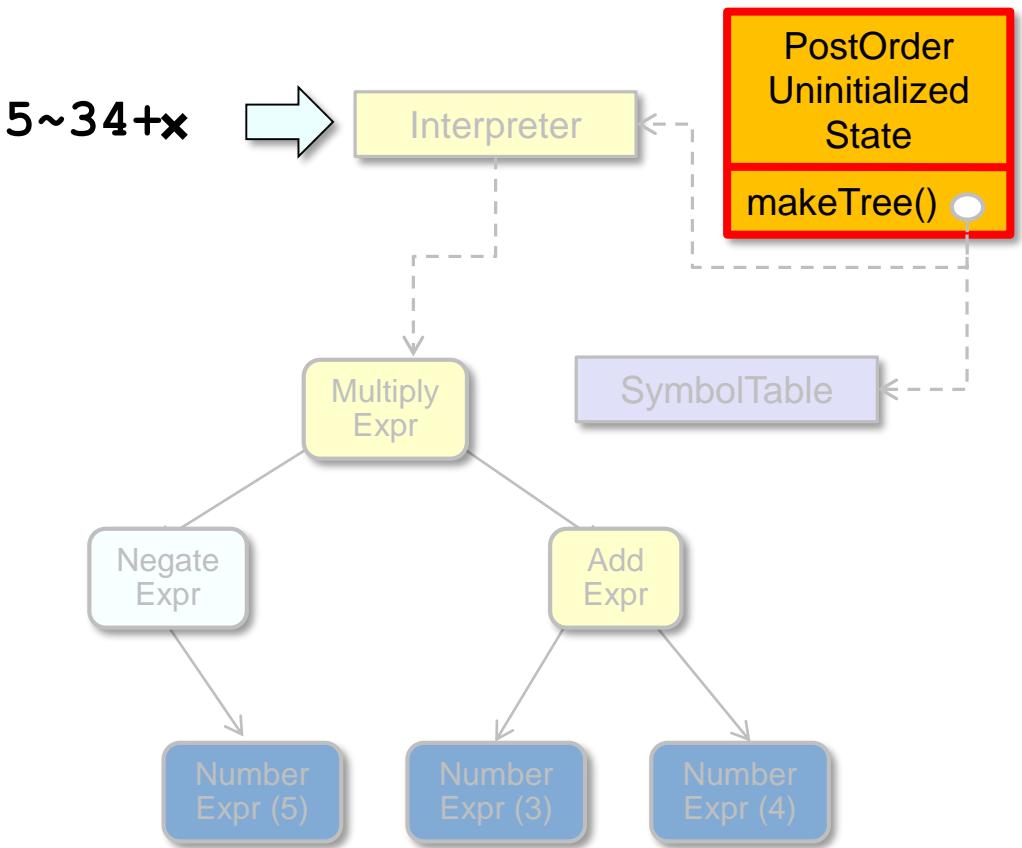
Solution: Create Interpreter to Process Input

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



Solution: Create Interpreter to Process Input

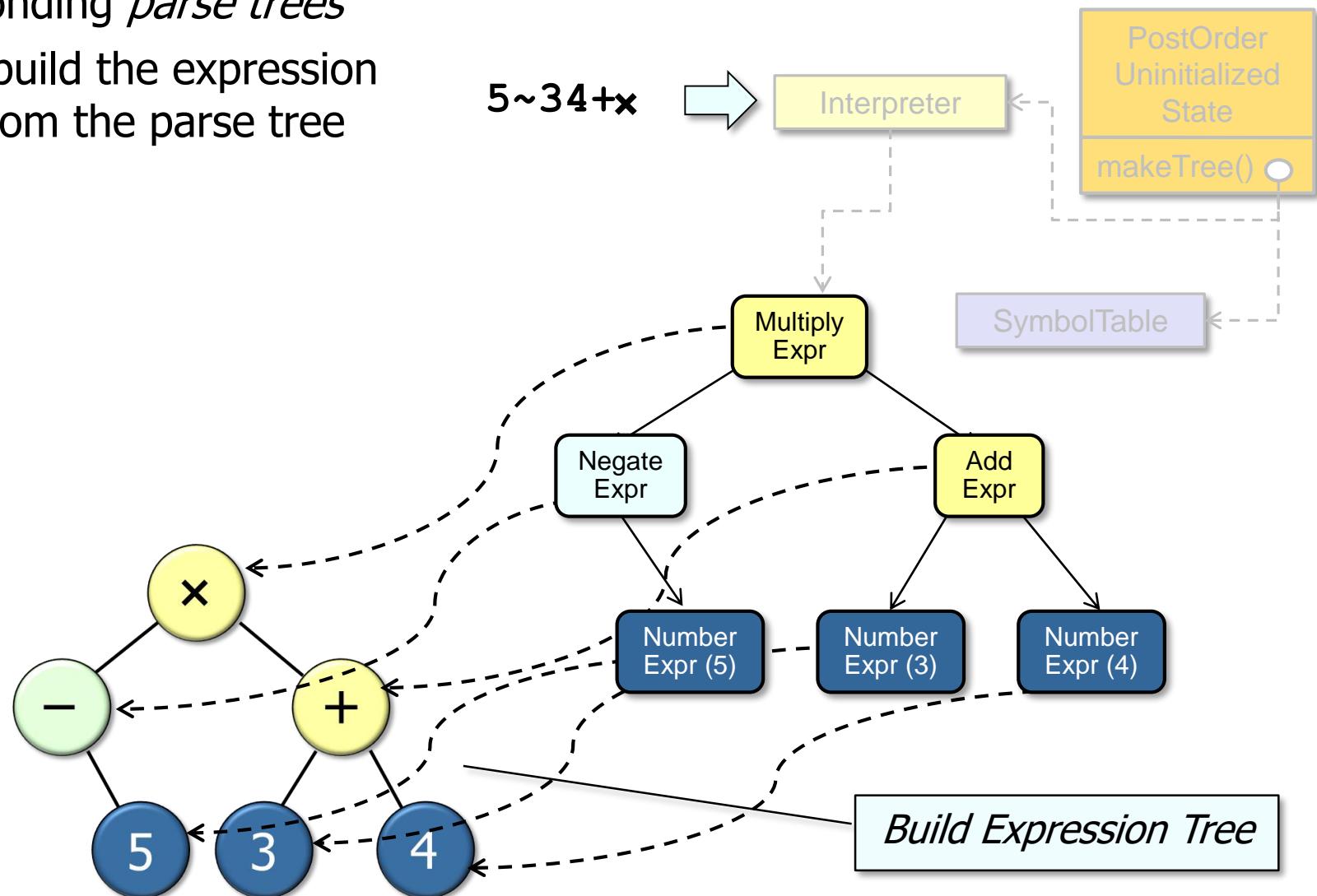
- Create an interpreter that processes user input expressions & creates the corresponding *parse trees*



The **PostOrderUninitializedState** class is covered in *State pattern* lesson

Solution: Create Interpreter to Process Input

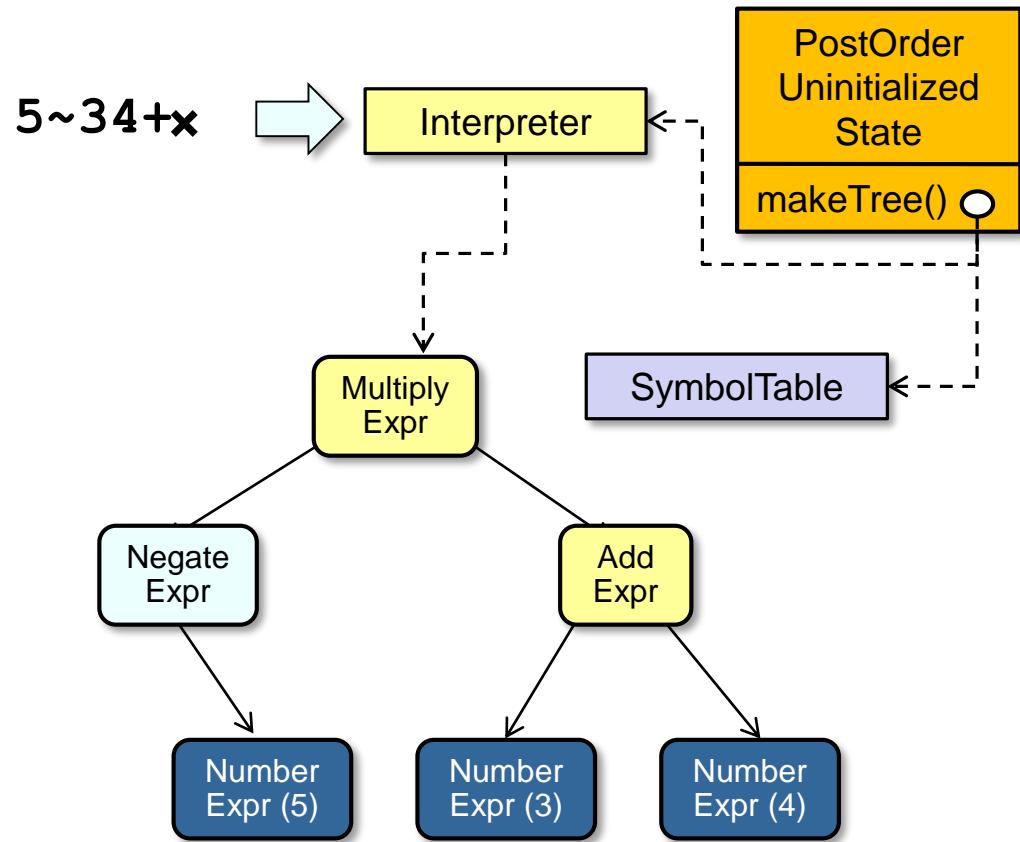
- Create an interpreter that processes user input expressions & creates the corresponding *parse trees*
 - Then build the expression tree from the parse tree



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 & creates the corresponding *parse trees*
 - Then build the expression tree from the parse tree

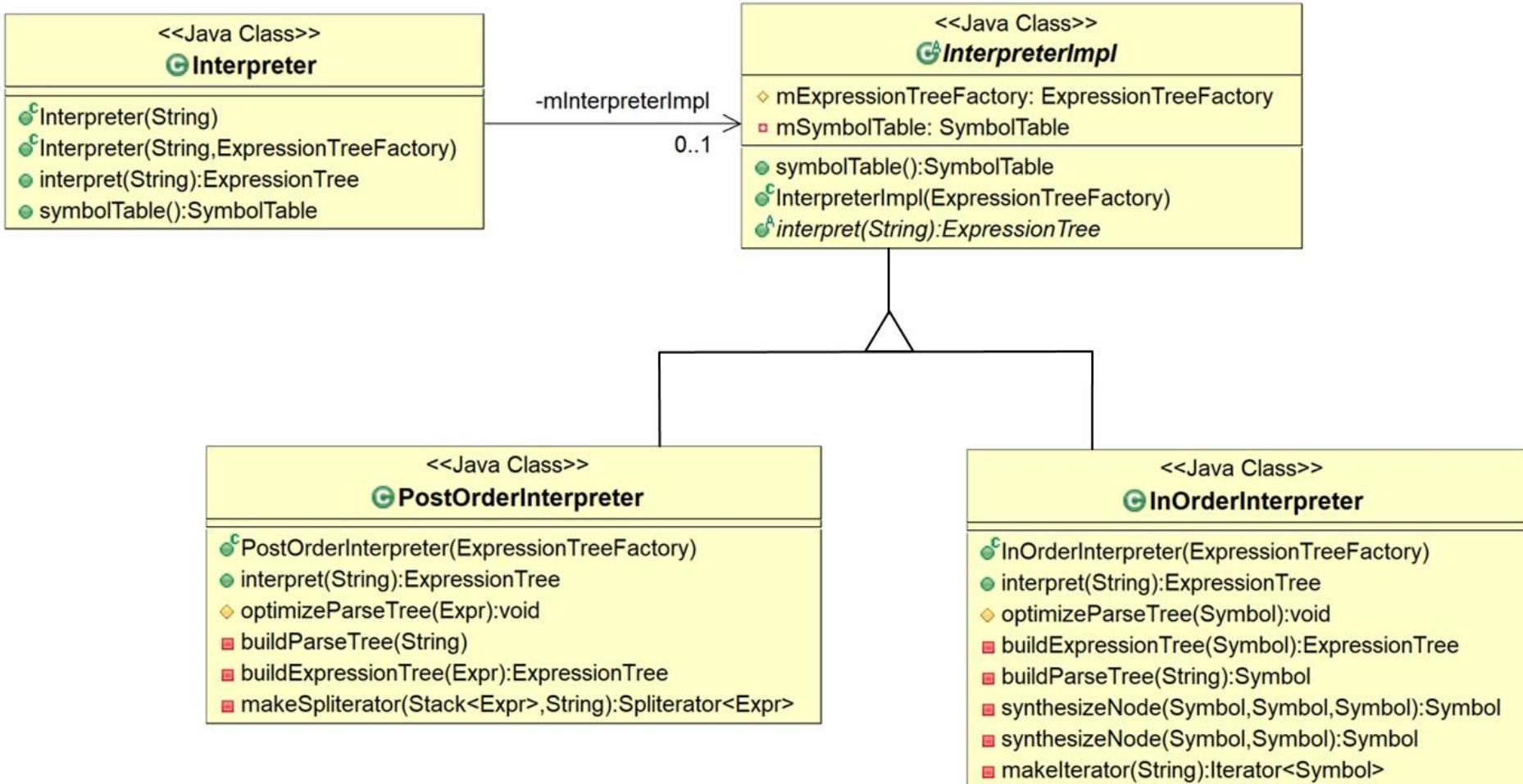


We just focus on using *Interpreter* to build the parse tree in this lesson

Interpreter Class Overview

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

Interpreter class hierarchy

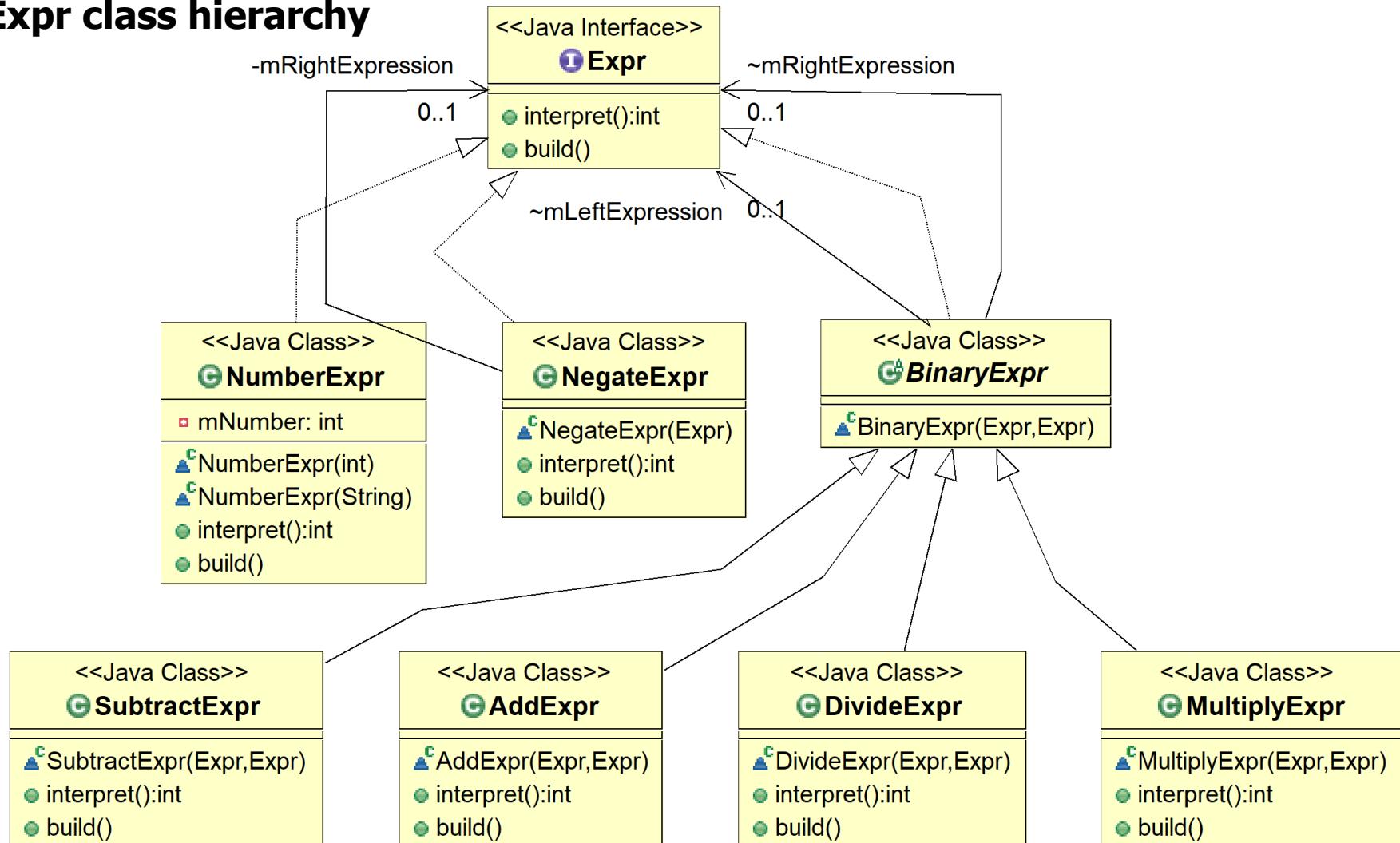


The *Bridge* pattern is used to support multiple interpreter implementations

Interpreter Class Overview

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

Expr class hierarchy



This class hierarchy includes portions of the *Interpreter* & *Builder* patterns

Interpreter Class Overview

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

Class methods

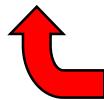
```
void optimizeParseTree()  
ExpressionTree buildExpressionTree()  
ExpressionTree interpret(String expression)
```

Interpreter Class Overview

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

Class methods

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

Hook methods
for optimization
& generation



Interpreter Class Overview

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

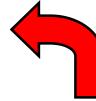
Class methods

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

```
SymbolTable()
```

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

<<creates>>



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

Interpreter Class Overview

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

Class methods

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

SymbolTable() <<creates>>

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

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

Elements of the Interpreter Pattern

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 ::= 'x' | '/'

add_sub ::= '+' | '-'

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

Applicability

- When the grammar is simple & relatively stable

```
expr ::= factor expr-tail

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

factor ::= term factor-tail

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

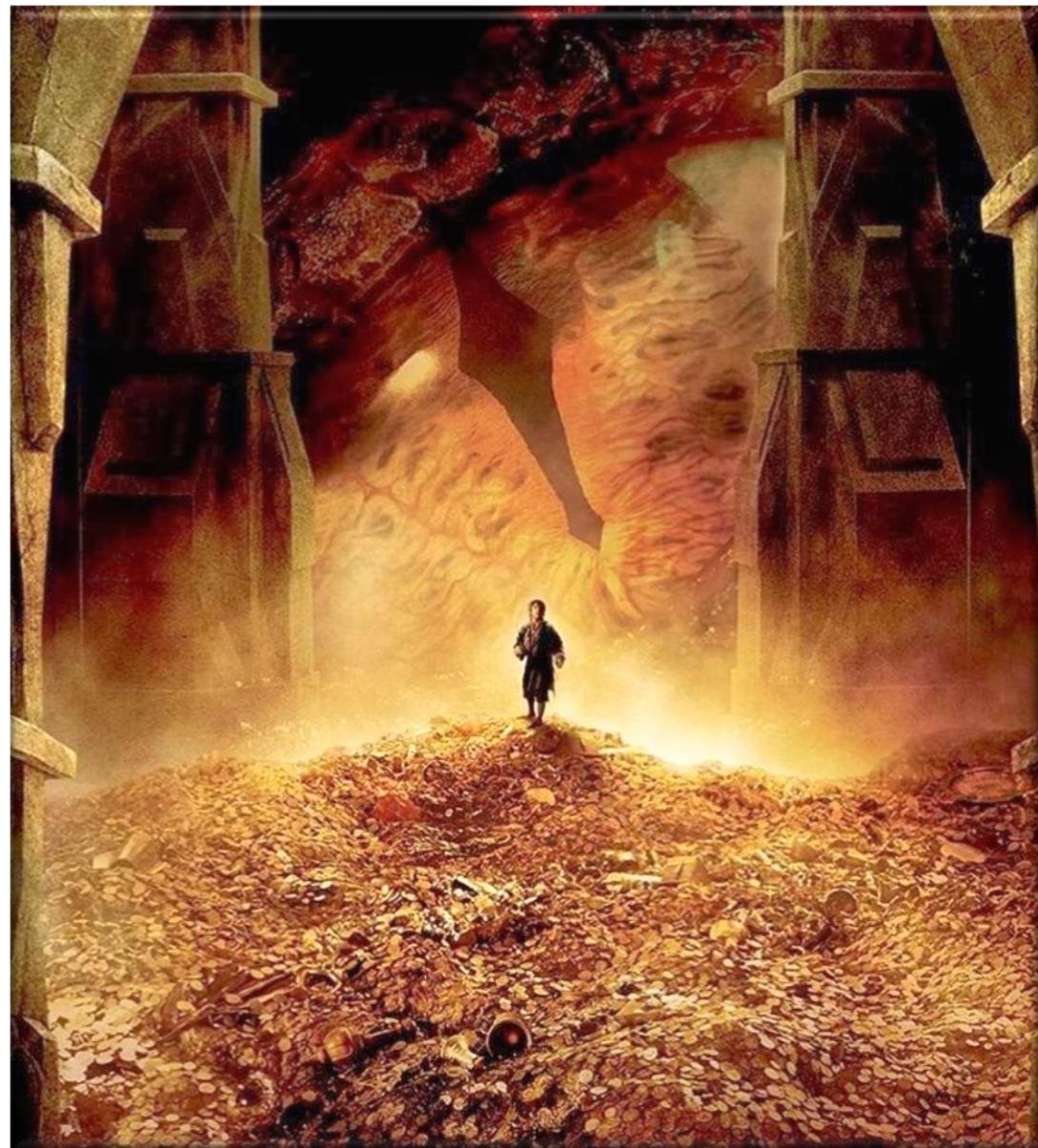
mul_div ::= 'x' | '/'

add_sub ::= '+' | '-'

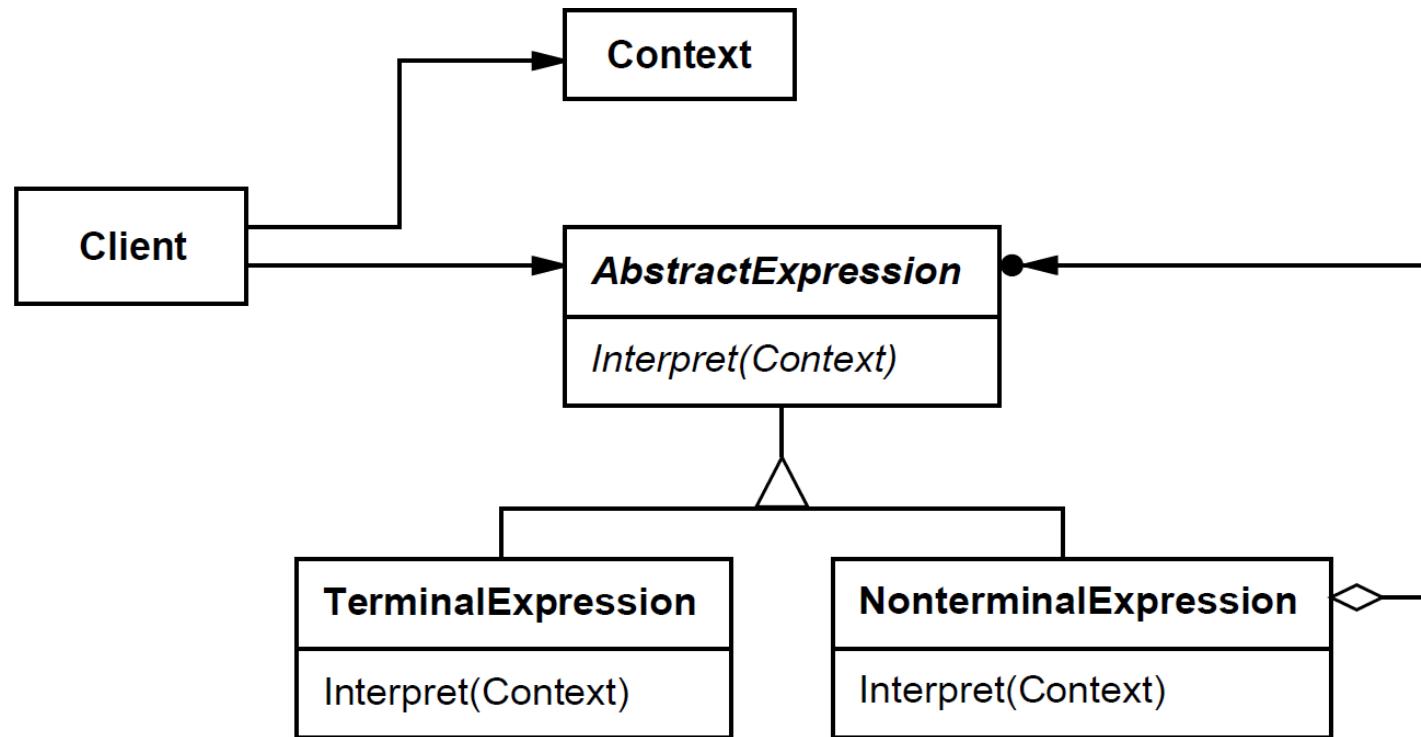
term ::= NUMBER | '(' expr ')' 
```

Applicability

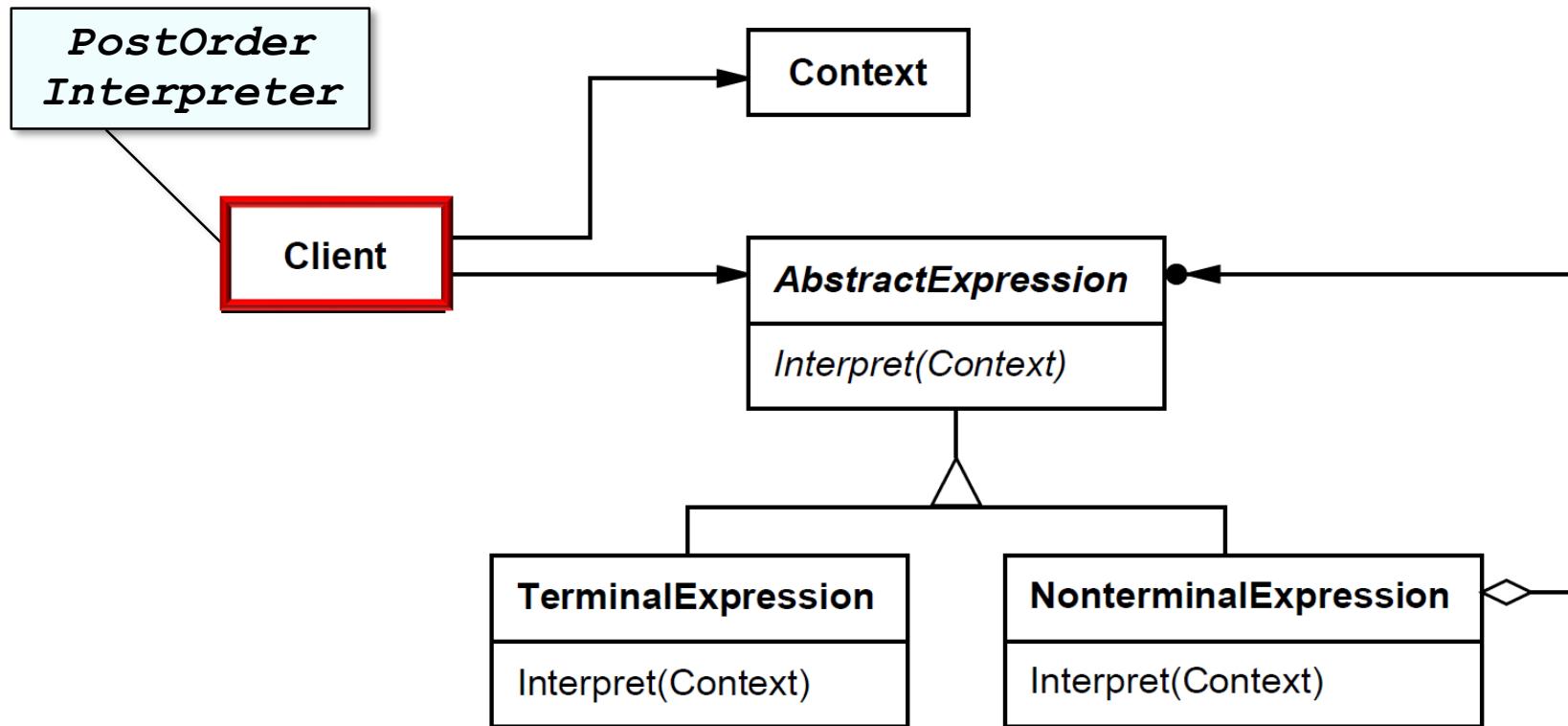
- When the grammar is simple & relatively stable
- When time/space efficiency is not a critical concern



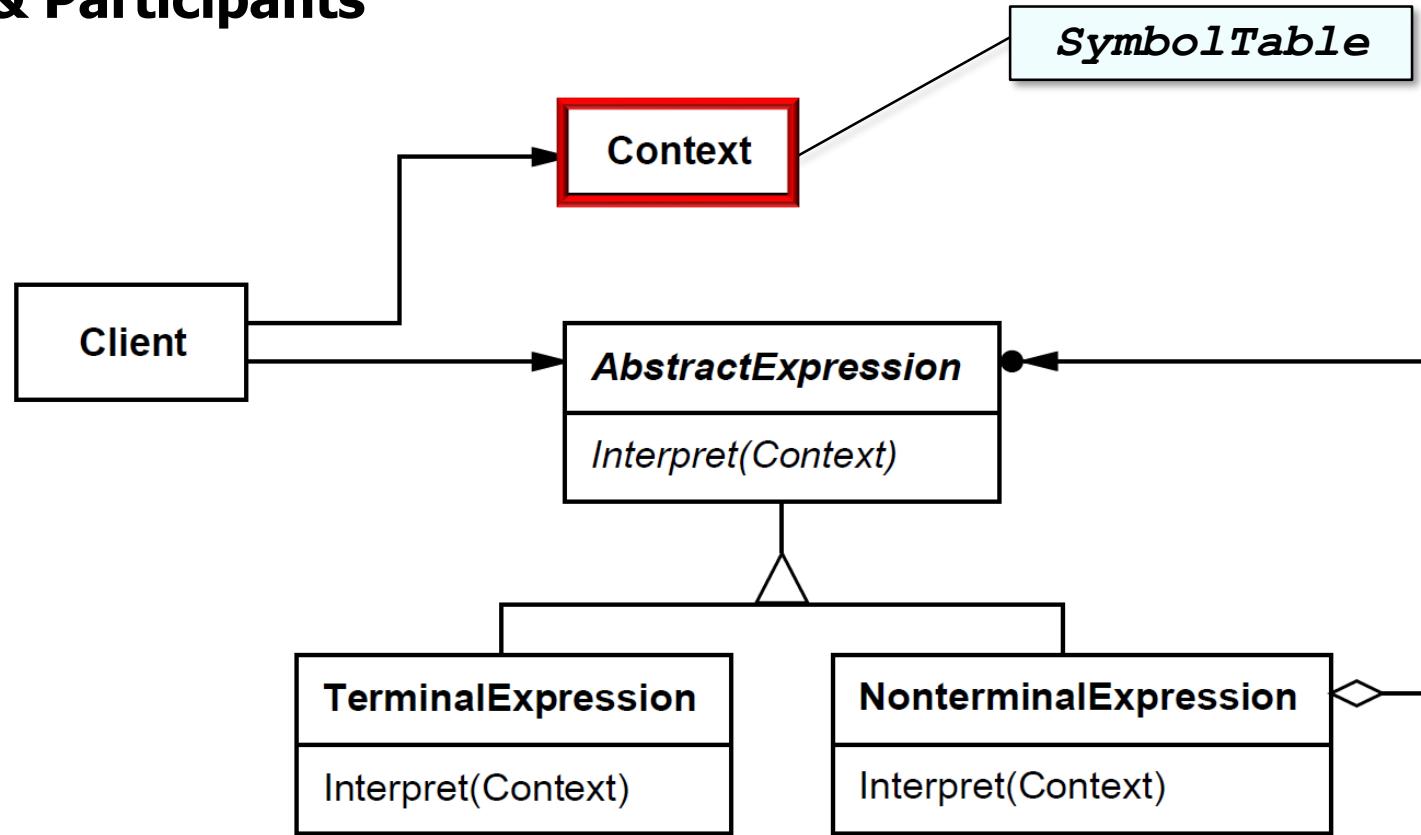
Structure & Participants



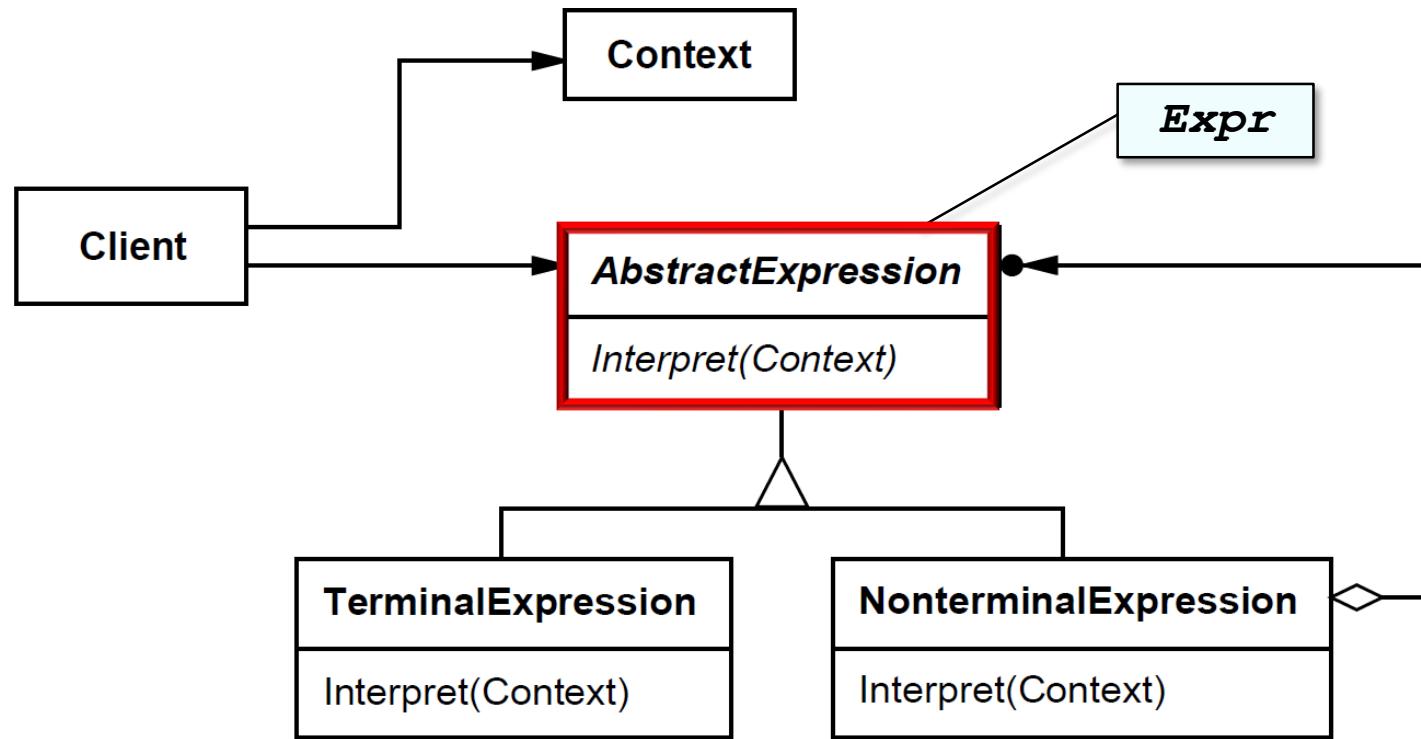
Structure & Participants



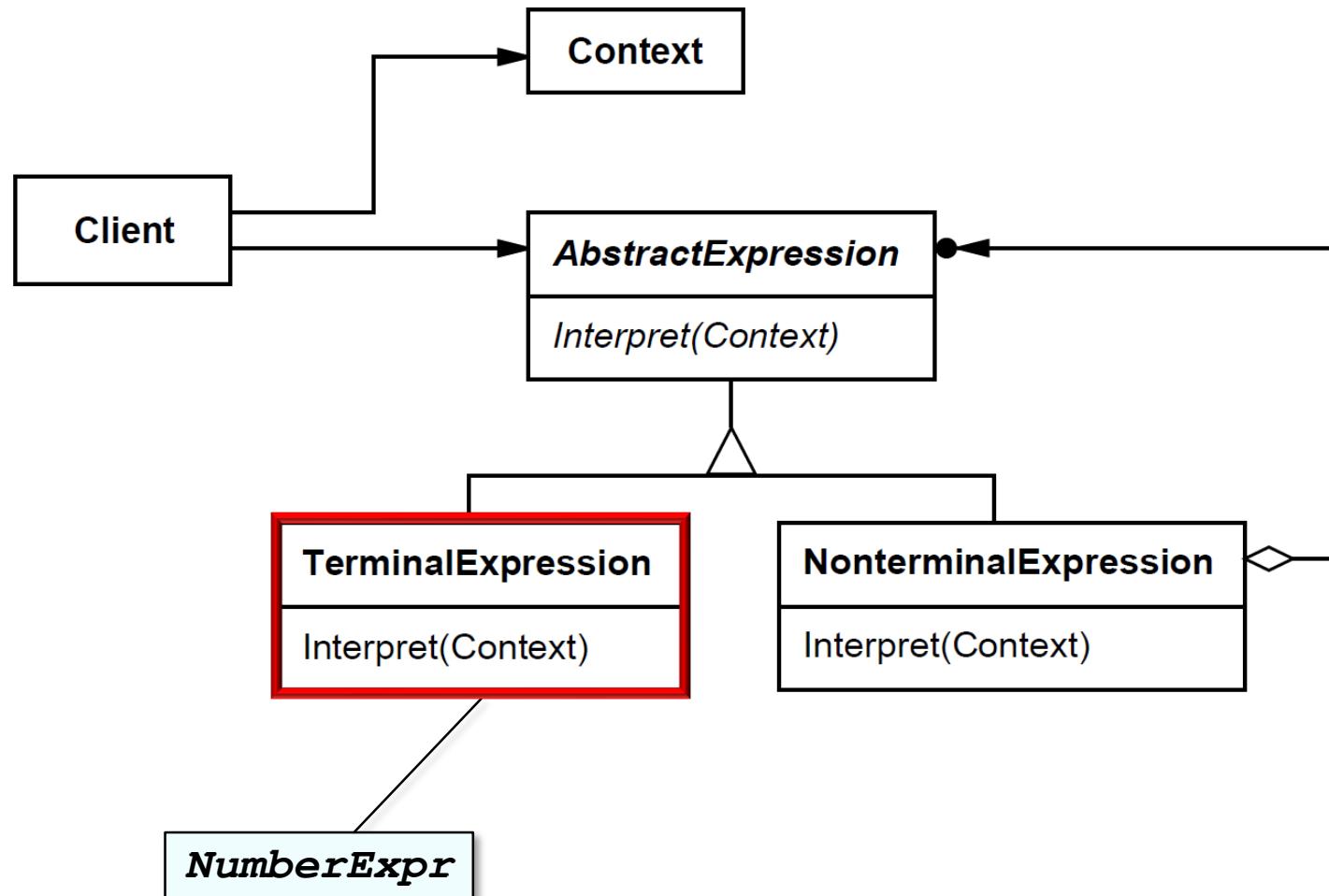
Structure & Participants



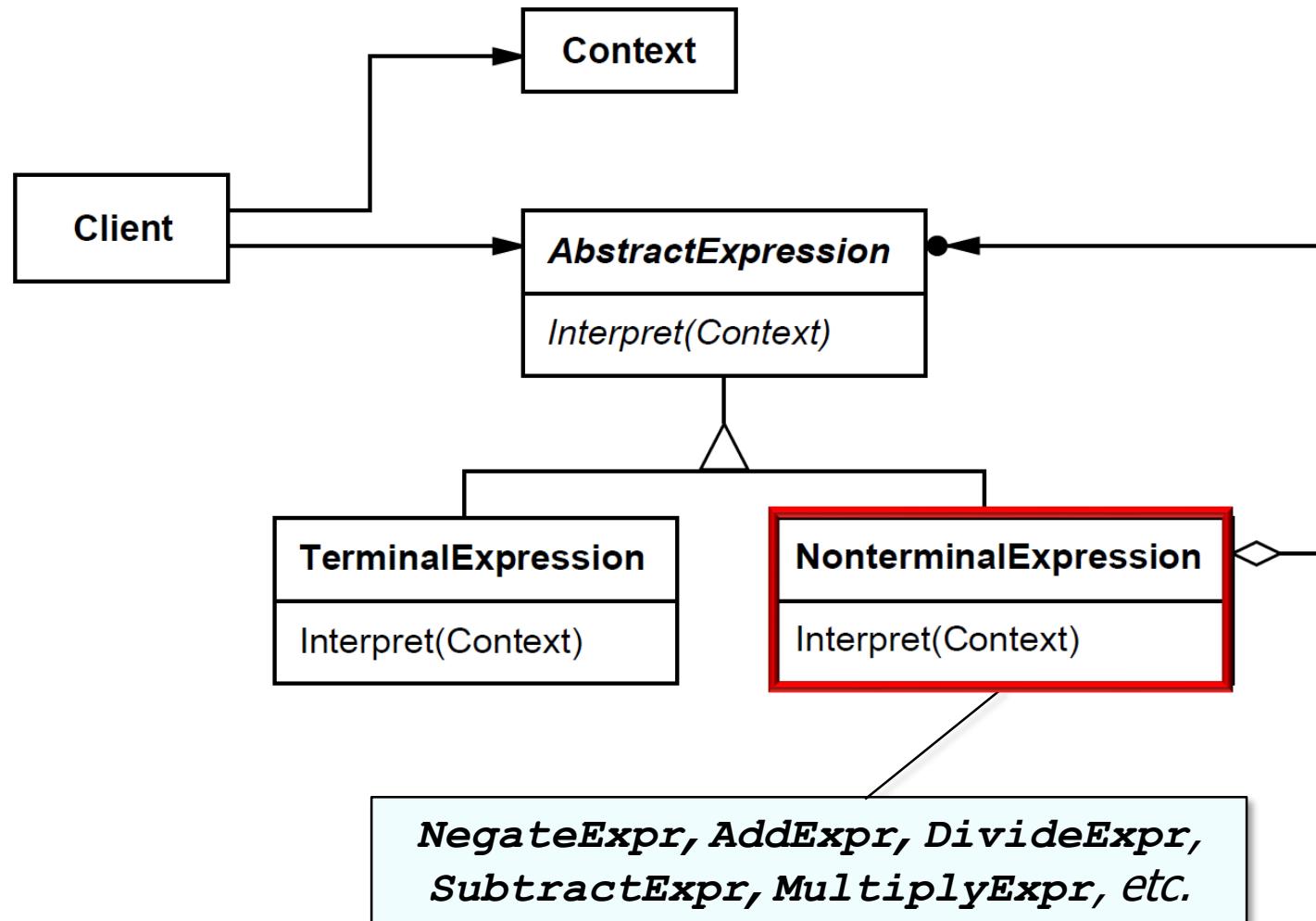
Structure & Participants



Structure & Participants



Structure & Participants



Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal 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;  
    }  
    ...  
}
```

Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal 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 terminal & non-terminal 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;  
    }  
    ...
```

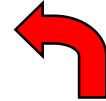


A parse tree interpreter/builder node
that handles # terminal expressions

Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal 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 # field

Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal 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 terminal & non-terminal expressions used to create a parse tree from user input

```
class NegateExpr implements Expr {    A parse tree interpreter/builder  
    private Expr mRightExpr;    ← that handles the unary minus  
                                operator non-terminal  
  
    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;  
    }  
}
```

Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal expressions used to create a parse tree from user input

```
class NegateExpr implements Expr {  
    private Expr mRightExpr; ← Constructor initializes the field  
  
    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;  
        }  
    }
```

Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal 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 non-terminal 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 terminal & non-terminal 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 operator
non-terminal expressions

Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal 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 terminal & non-terminal expressions used to create a parse tree from user input

```
class MultiplyExpr extends BinaryExpr {
```

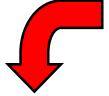


A parse tree interpreter/builder that handles multiply operator non-terminal expressions

```
    MultiplyExpr(Expr leftExpression,  
                 Expr rightExpression) {  
        super(leftExpression, rightExpression);  
    }  
  
    public int interpret() {  
        return mLeftExpression.interpret()  
            * mRightExpression.interpret();  
    }
```

Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal expressions used to create a parse tree from user input

```
class MultiplyExpr extends BinaryExpr {  
  
     Constructor initializes the  
    superclass fields  
  
    MultiplyExpr(Expr leftExpression,  
                 Expr rightExpression) {  
        super(leftExpression, rightExpression);  
    }  
  
    public int interpret() {  
        return mLeftExpression.interpret()  
            * mRightExpression.interpret();  
    }  
}
```

Interpreter example in Java

- The `Expr` hierarchy defines terminal & non-terminal 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 non-terminal by multiplying stored expressions

Interpreter example in Java

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

```
public 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

```
public class PostOrderInterpreter {  
    ...  
    ExpressionTree interpret(String inputExpression) {  
        Expr parseTree = buildParseTree(inputExpression);  
  
        Parse input & build parseTree  
        ↓  
        if (!parseTree.isEmpty()) {  
  
            optimizeParseTree(parseTree);  
  
            return buildExpressionTree(parseTree);  
        }  
        ...  
    }
```

Interpreter example in Java

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

```
public 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

```
public 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

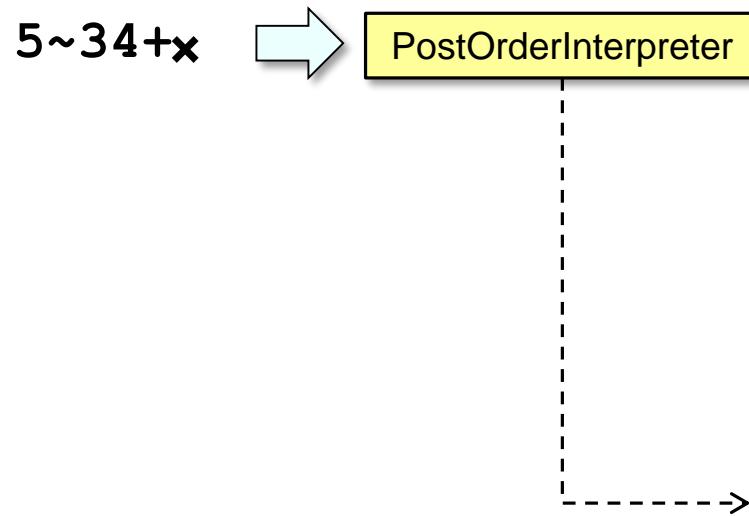
```
public class PostOrderInterpreter {  
    ...  
    ExpressionTree interpret(String inputExpression) {  
        Expr parseTree = buildParseTree(inputExpression);  
  
        if (!parseTree.isEmpty()) {  
  
            optimizeParseTree(parseTree);  
  
            return buildExpressionTree(parseTree);  
        }  
        ...  
    }
```

Template method

Hook methods (primitive operations)

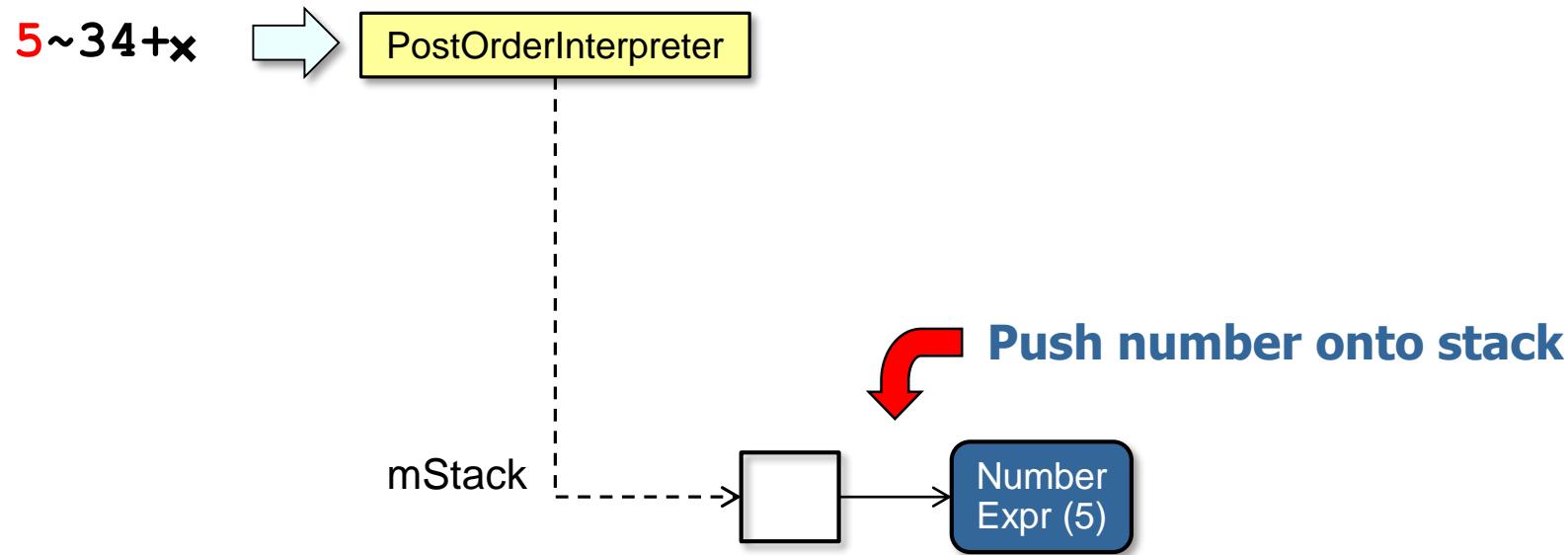
Interpreter example in Java

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



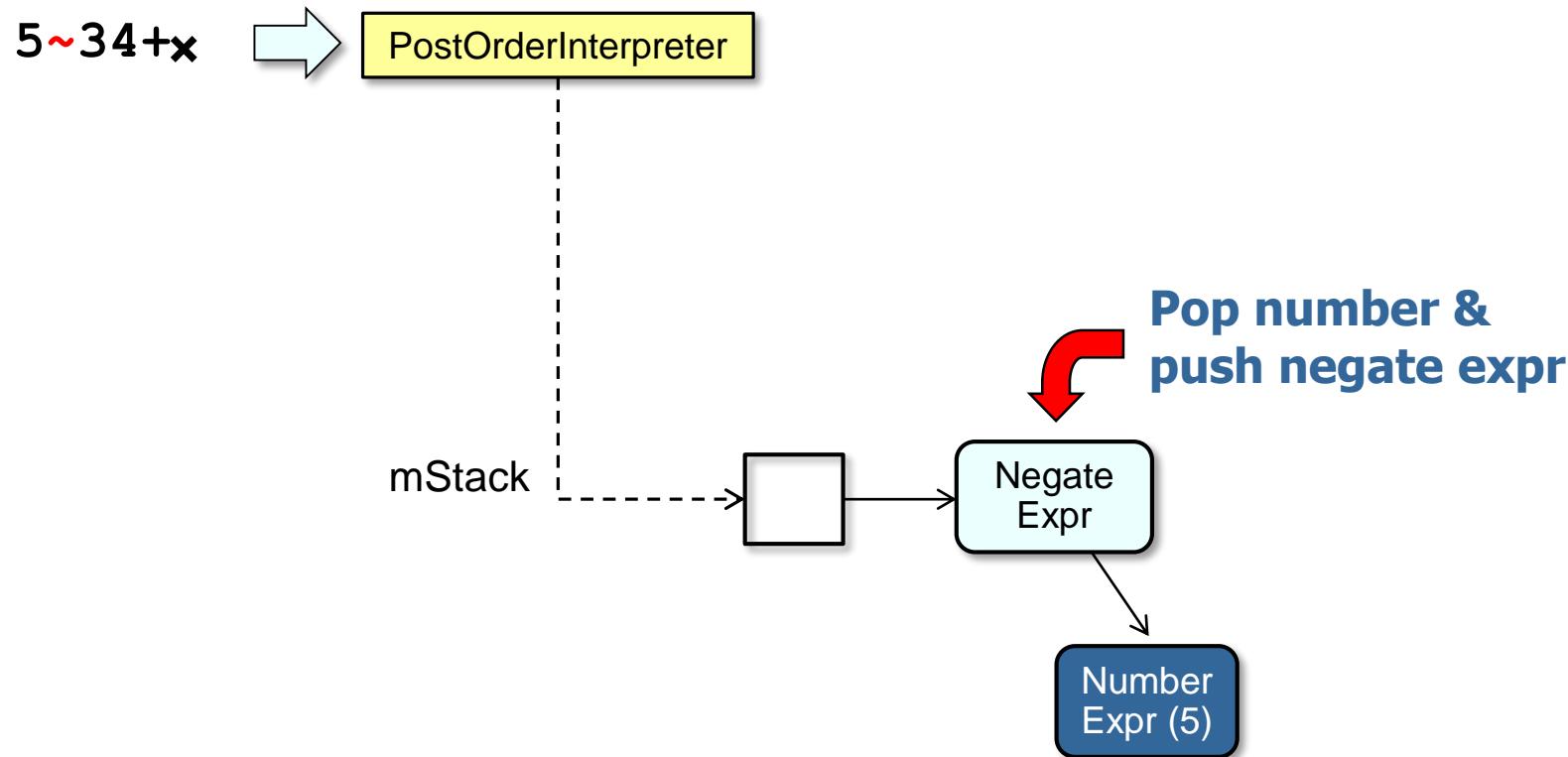
Interpreter example in Java

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



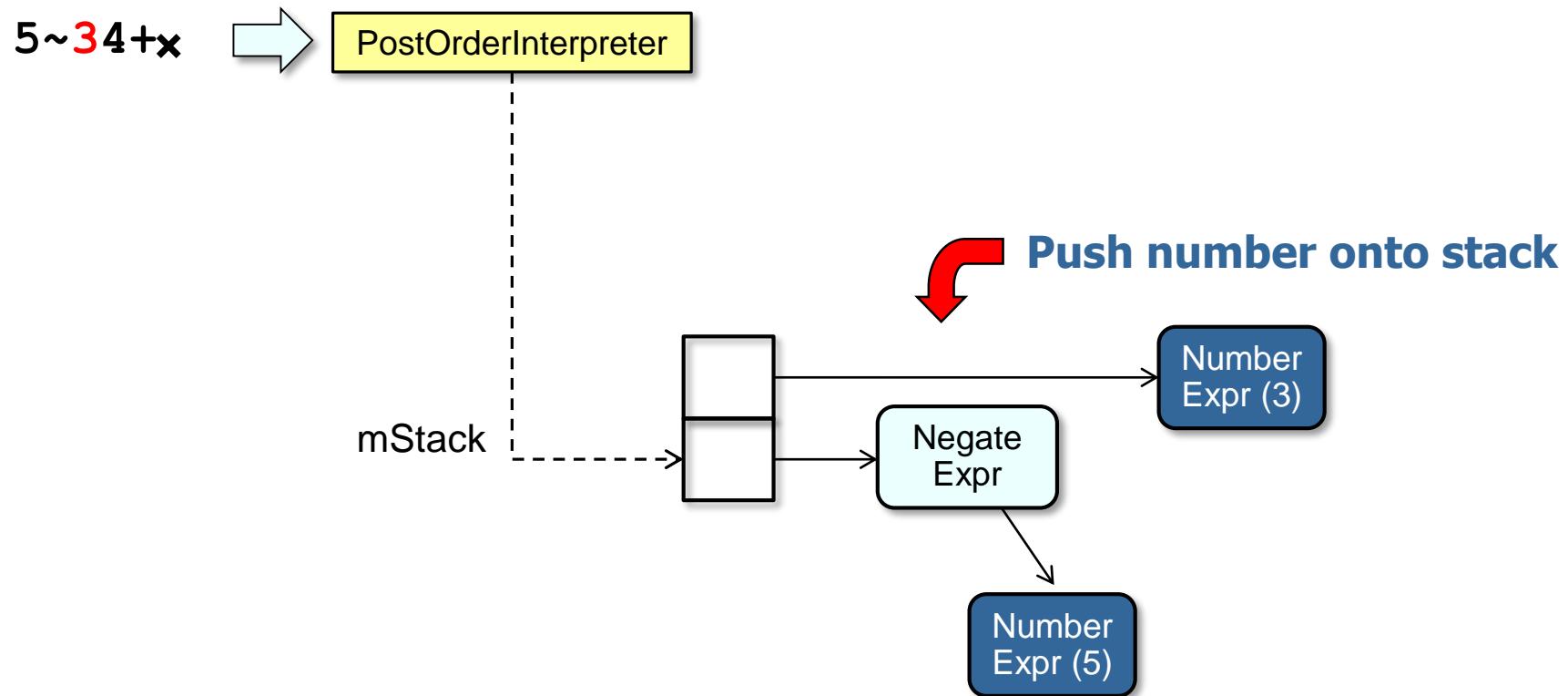
Interpreter example in Java

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



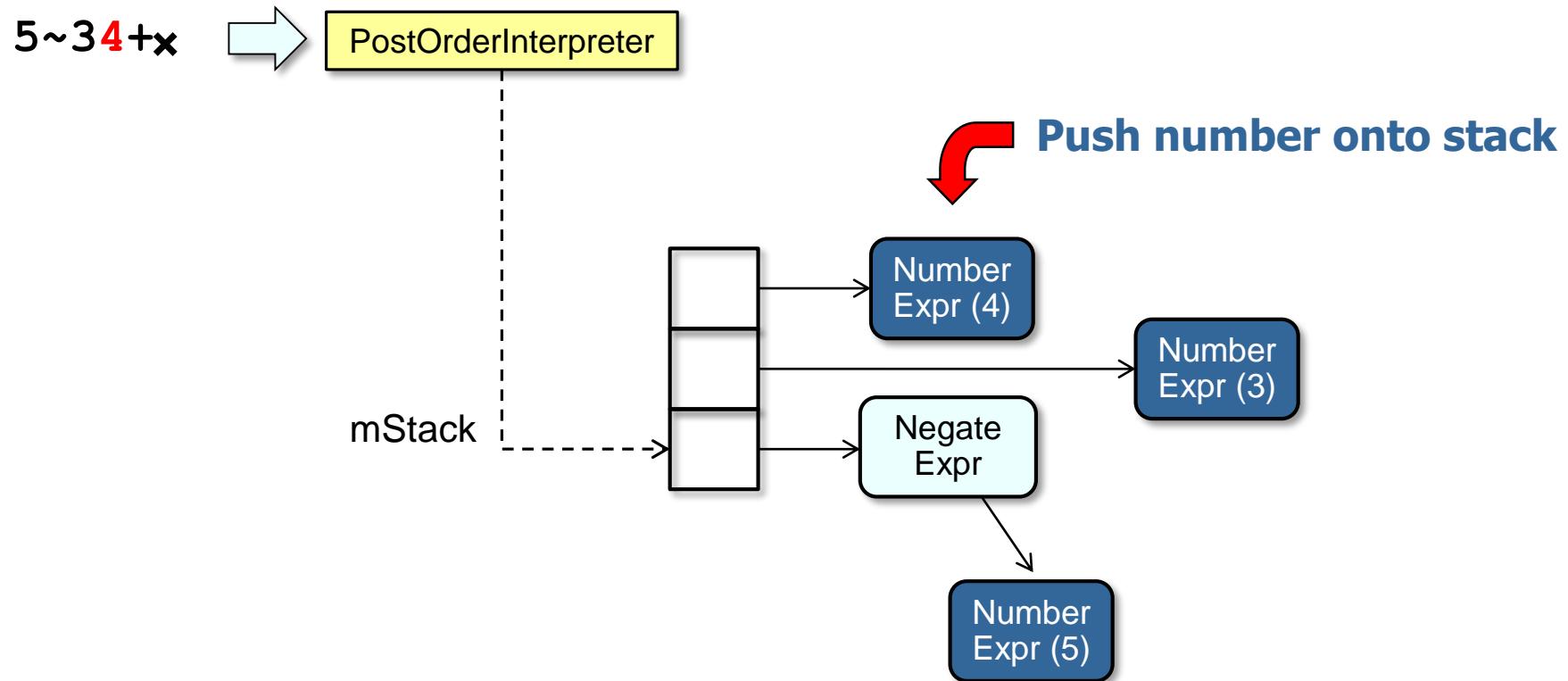
Interpreter example in Java

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



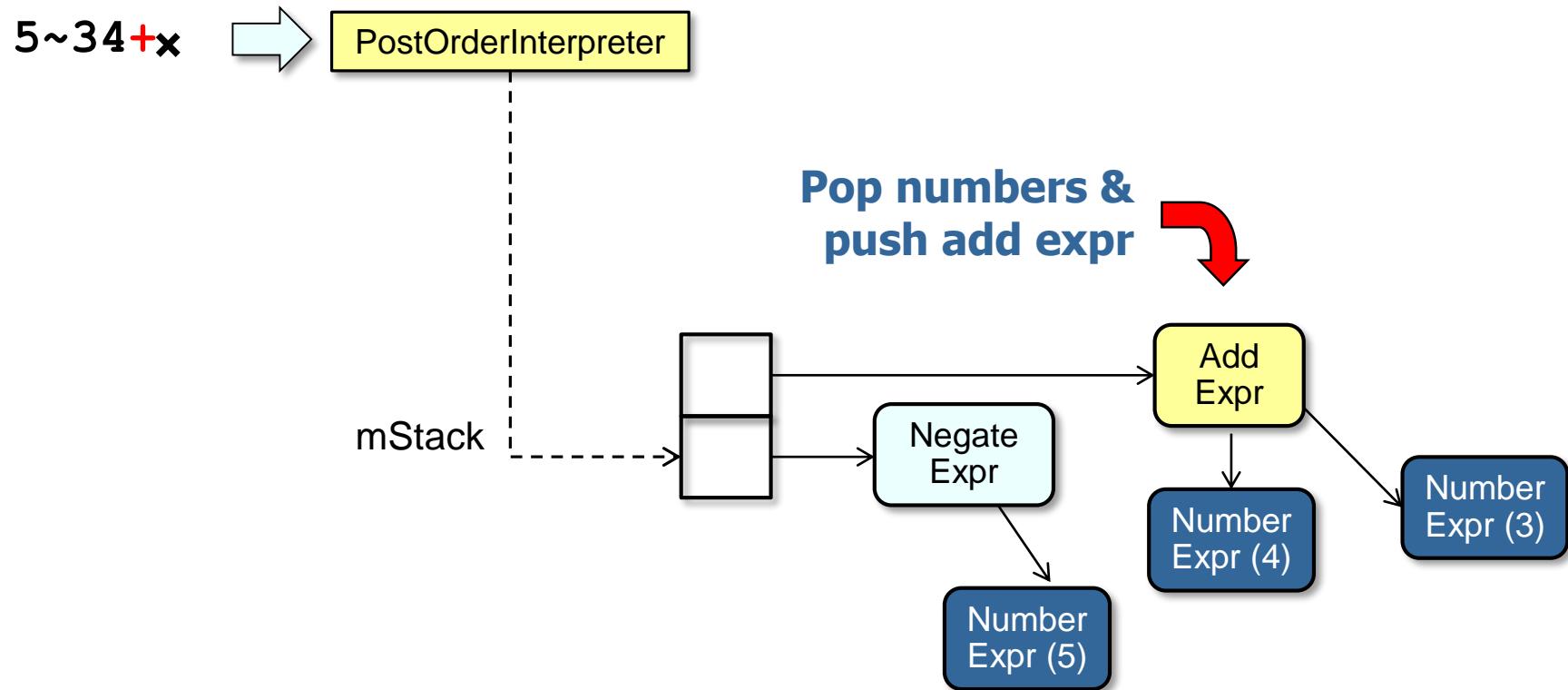
Interpreter example in Java

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



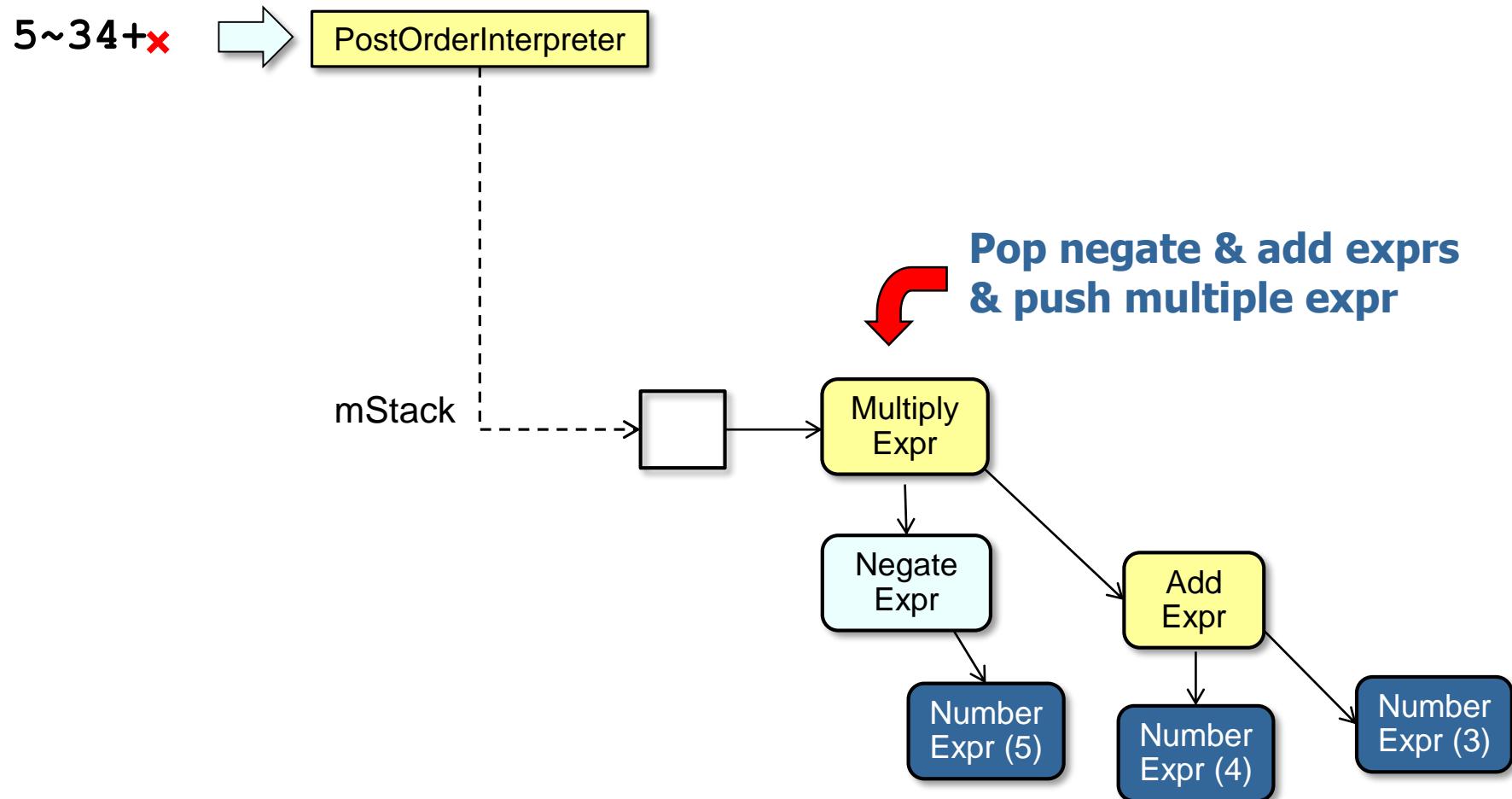
Interpreter example in Java

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



Interpreter example in Java

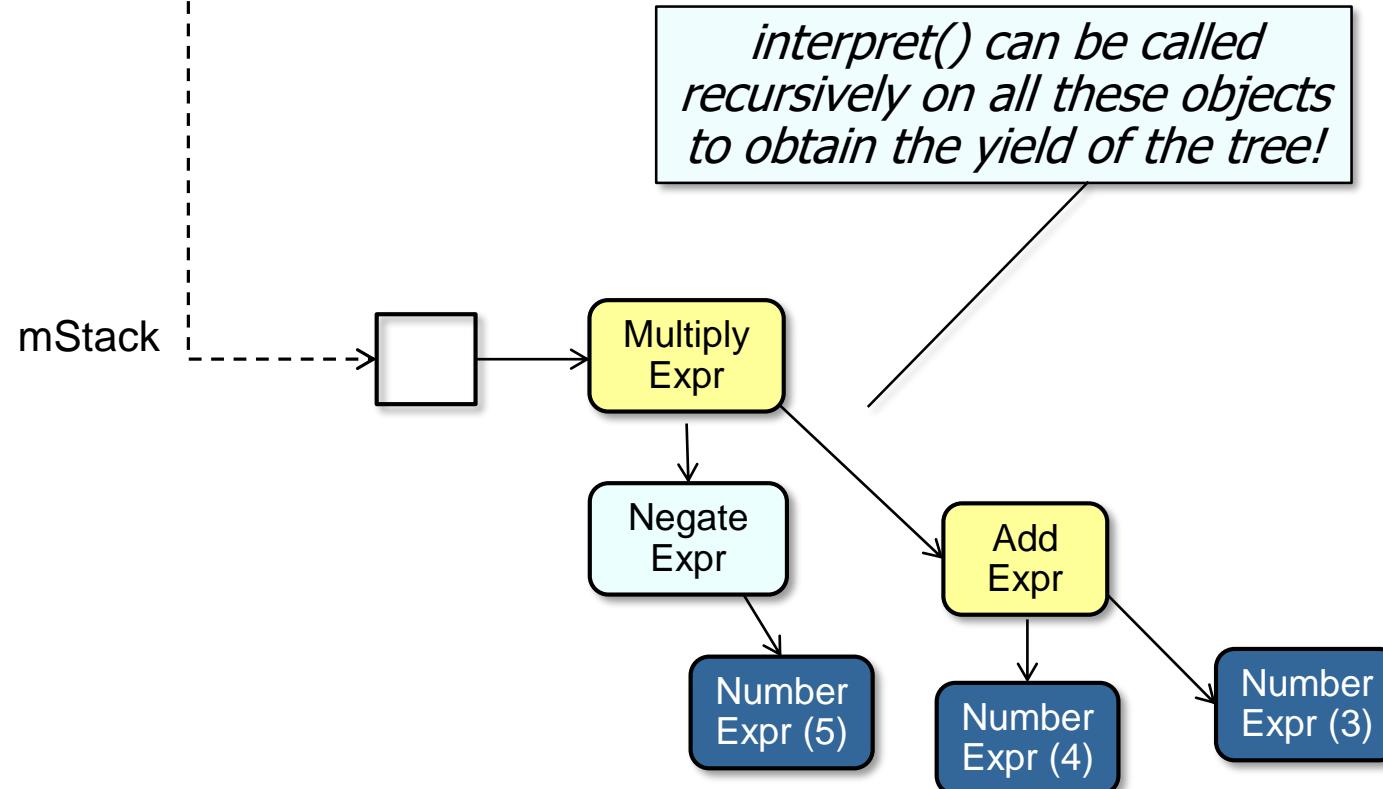
- `ExprSpliterator.tryAdvance()` processes the next symbol in the user's input expression



Interpreter example in Java

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

5~34+x → PostOrderInterpreter



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
            ...
    }
}
```

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

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++);
            
            Get the next input character
            while (Character.isWhitespace(c))
                c = mInputExpression.charAt(mIndex++);
            ...
            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++);

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

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 '+': ← Handle the addition operator
                        mStack.push(new AddExpr(mStack.pop(), rightExpr));
                        break;
                    case '-':
                        mStack.push(new SubtractExpr(mStack.pop(),
                            rightExpr));
                        break;
                    ...
                }
            }
        }
    }
}
```

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 '-': ← Handle the subtraction operator
                        mStack.push(new SubtractExpr(mStack.pop(),
                            rightExpr));
                        break;
                    ...
                }
            }
        }
    }
}
```

Consequences

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

```
expr ::= factor expr-tail

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

factor ::= term factor-tail

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

mul_div ::= 'x' | '/'

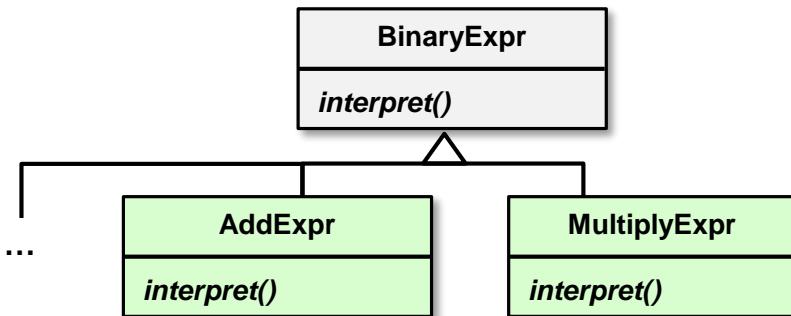
add_sub ::= '+' | '-'

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

This grammar removes immediate left recursion

Consequences

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



```
expr ::= factor expr-tail
expr-tail ::= add_sub expr
            | /* empty */;
factor ::= term factor-tail
factor-tail ::= mul_div factor
              | /* empty */;
mul_div ::= 'x' | '/';
add_sub ::= '+' | '-';
term ::= NUMBER | '(' expr ')' ;
```

Consequences

- Complex grammars hard to create & maintain
 - e.g., more rules that are inter-dependent yield more inter-dependent classes

```
postfix-expression ::=  
primary-expression  
postfix-expression [ expression ]  
postfix-expression ( expression-listopt )  
simple-type-specifier ( expression-listopt )  
typename::opt nested-name-  
specifier identifier ( expression-listopt )  
typename::opt nested-name-specifier templateopt  
template-id ( expression-listopt )  
postfix-expression . templateopt id-expression  
postfix-expression -> templateopt 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 hard to create & maintain
 - e.g., more rules that are inter-dependent yield more inter-dependent classes

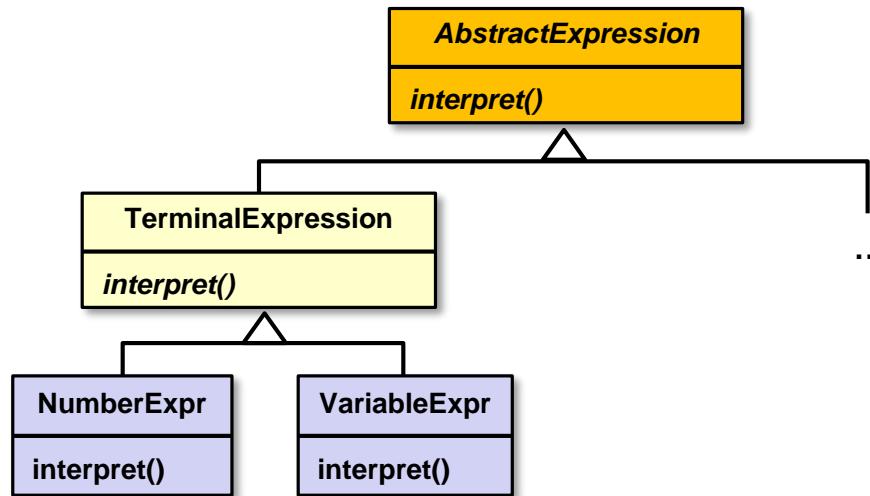
*Complex grammars often require different approach,
e.g., parser generators*



Javacc™

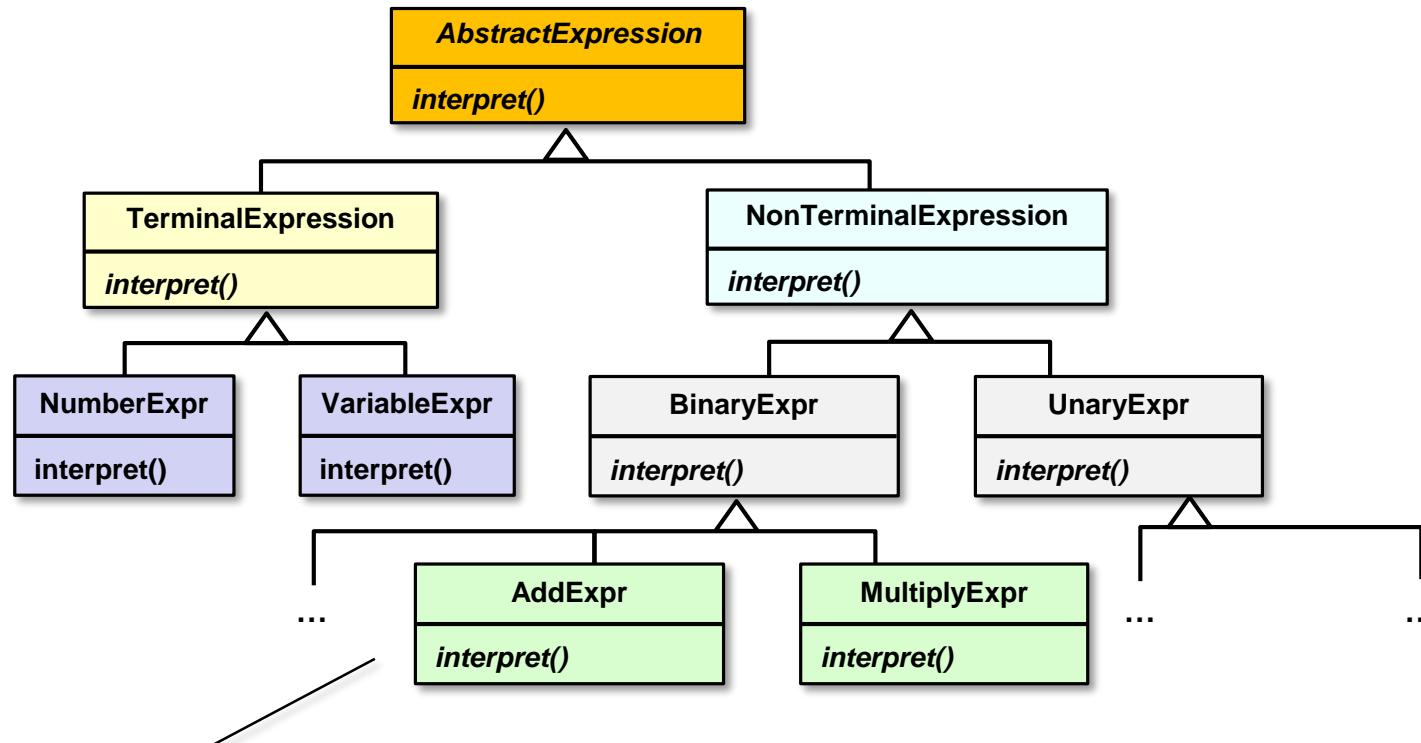
Implementation

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



Implementation

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



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

Implementation

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

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

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

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

-5x (3+4)

InOrderInterpreter
interpret()
createParseTree()
optimizeParseTree()
buildExpressionTree()

Implementation

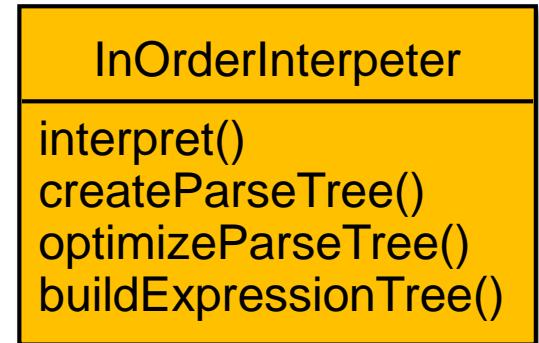
- Various approaches
 - Express language rules, one per class
 - Use operator precedence for an interpreter that builds parse trees from in-order expressions

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

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

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

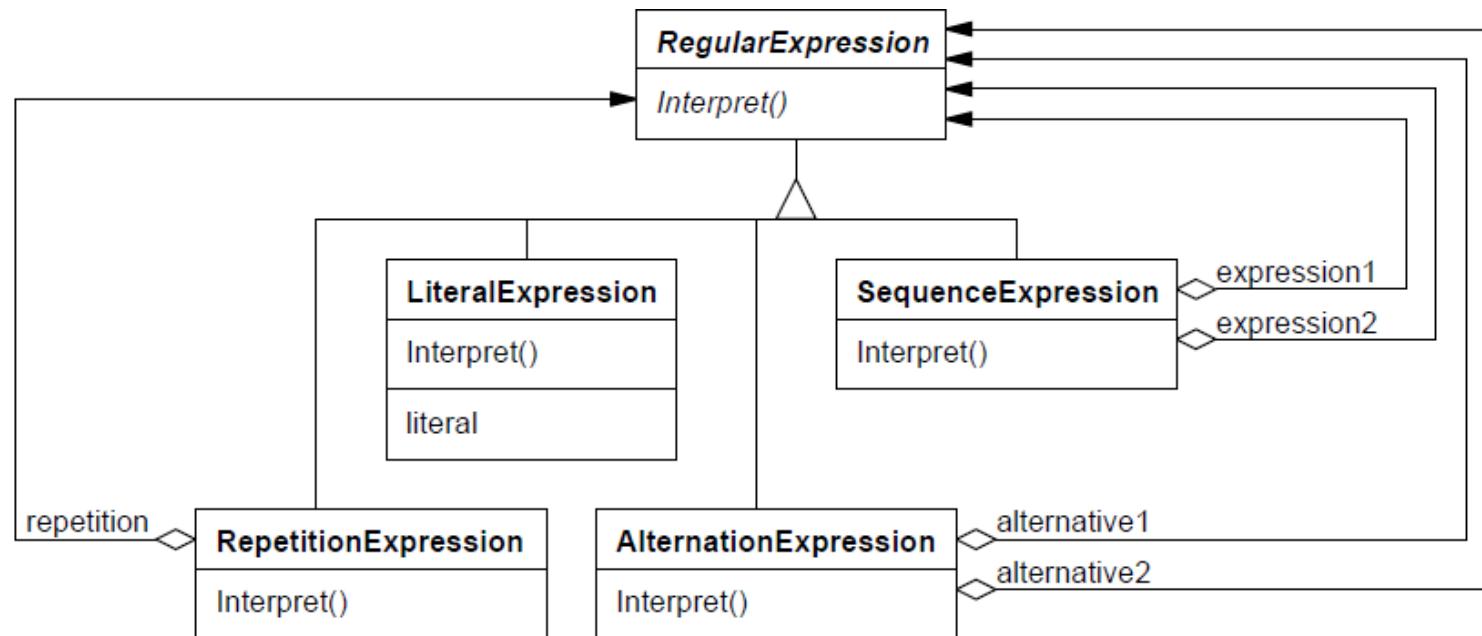
-5x (3+4)



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

Known Uses

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



Known Uses

- Text editors & web browsers use the *Interpreter* pattern to lay out documents & check spelling
 - e.g., an equation in TeX is represented as a tree where internal nodes are operators & 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`

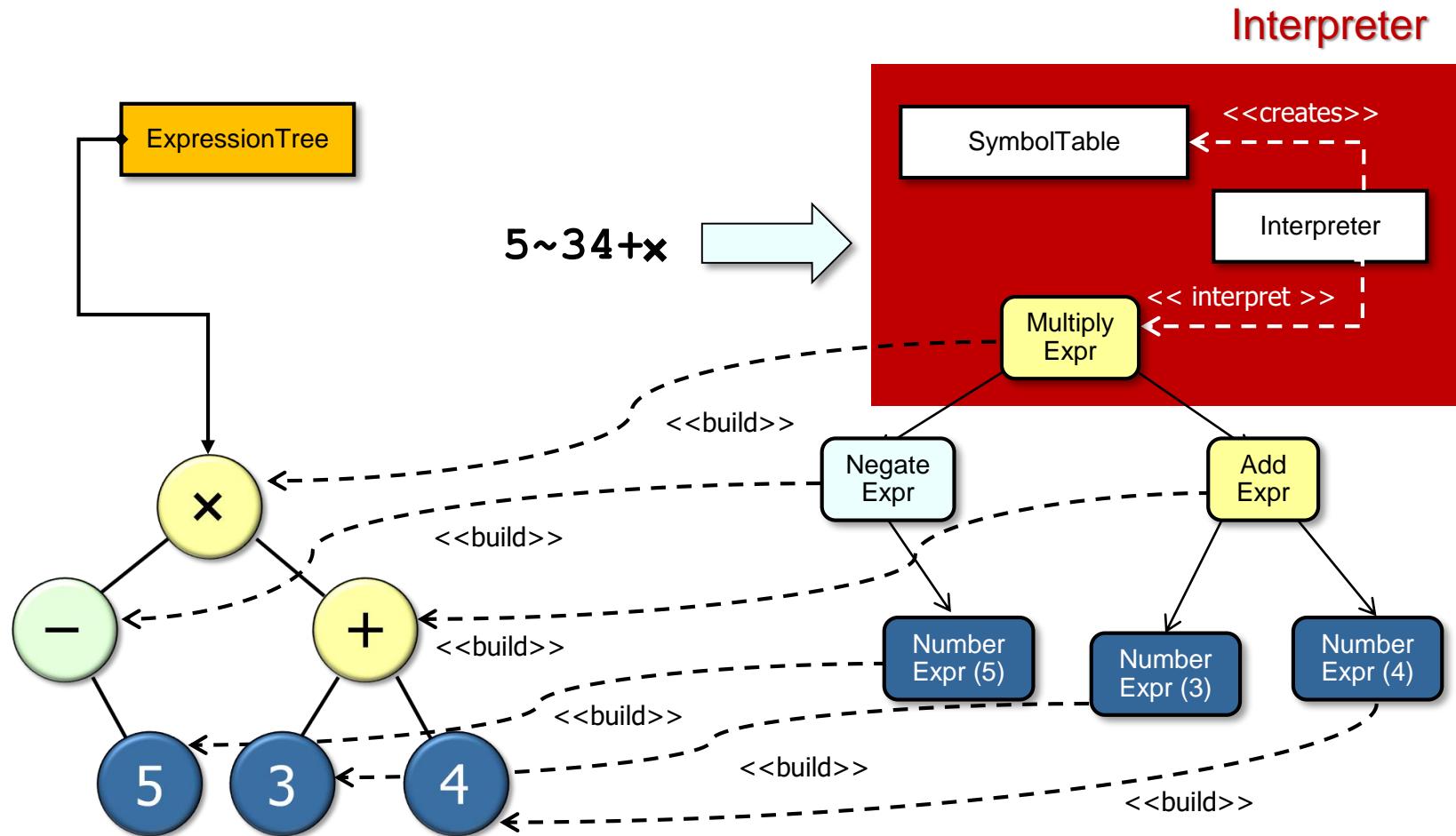
```
public final class Pattern
extends Object
implements Serializable
```

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.

Summary of the Interpreter Pattern

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



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

End of the
Interpreter Pattern

The Builder Pattern

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software
Integrated Systems

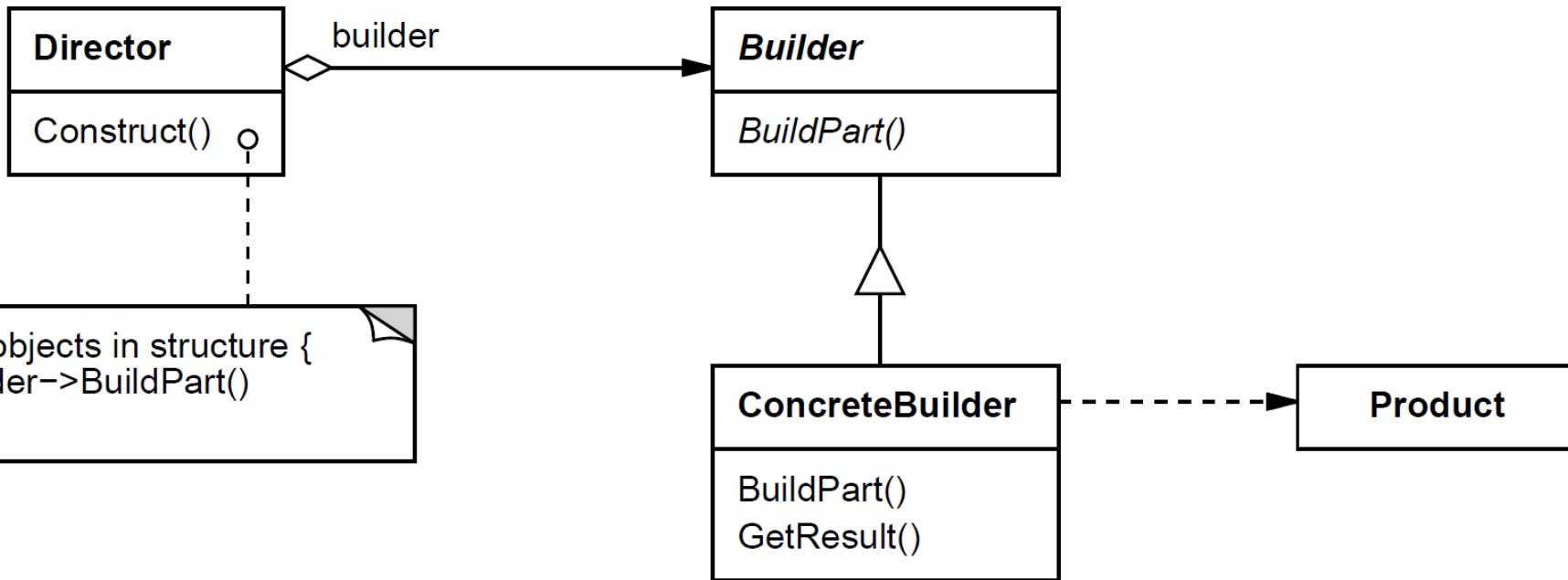
Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives

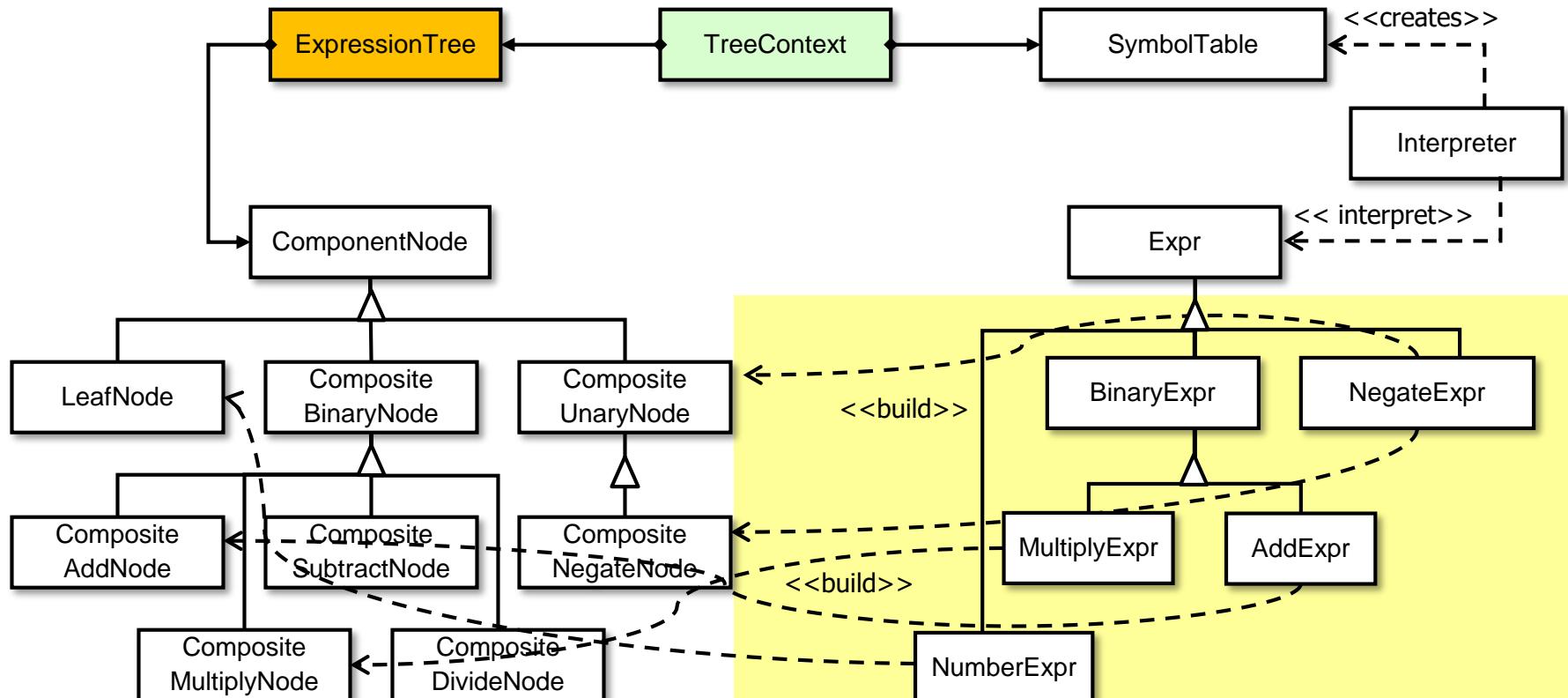
Learning Objectives

- Understand the *Builder* pattern



Learning Objectives

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

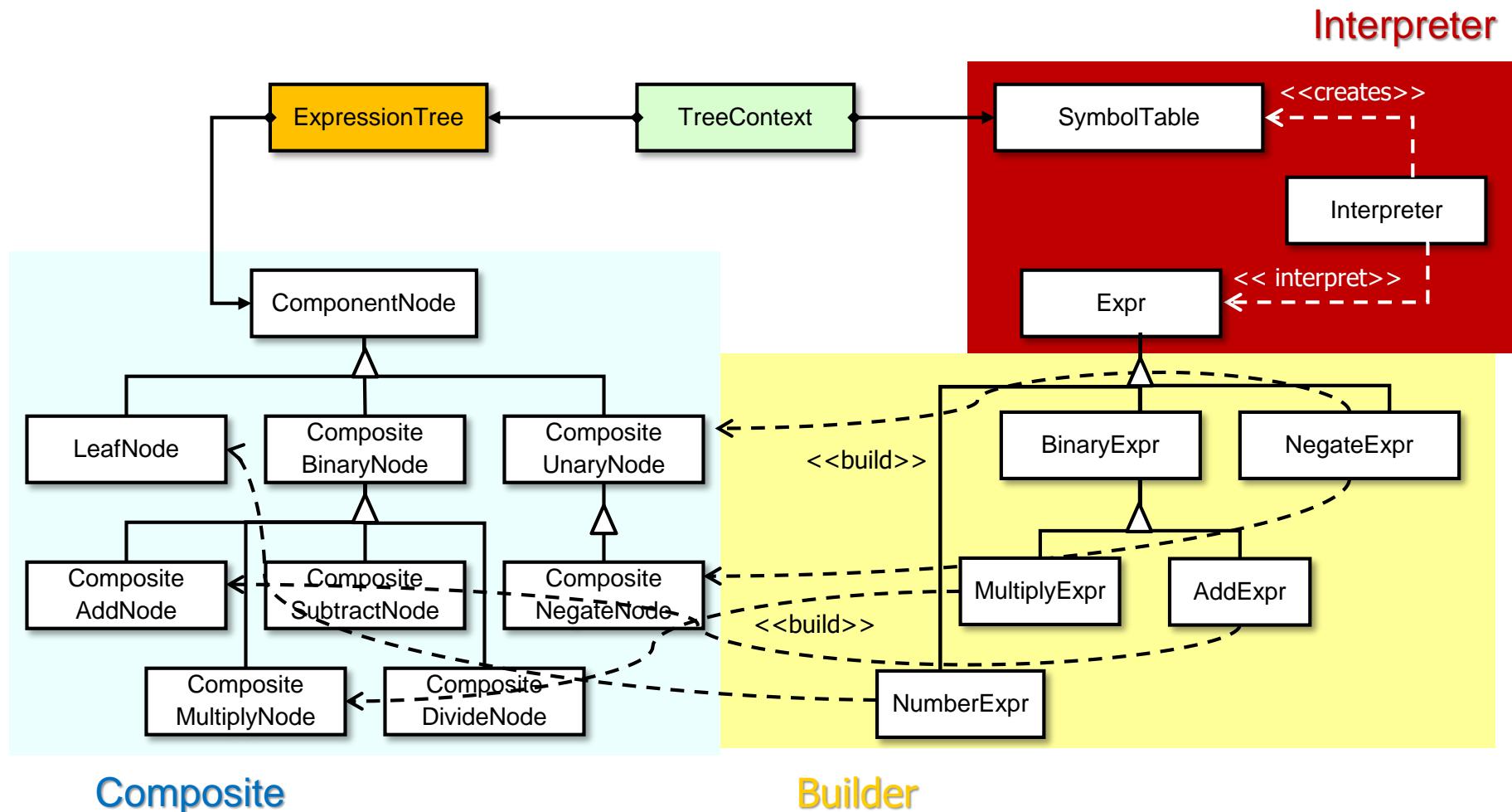


Builder

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

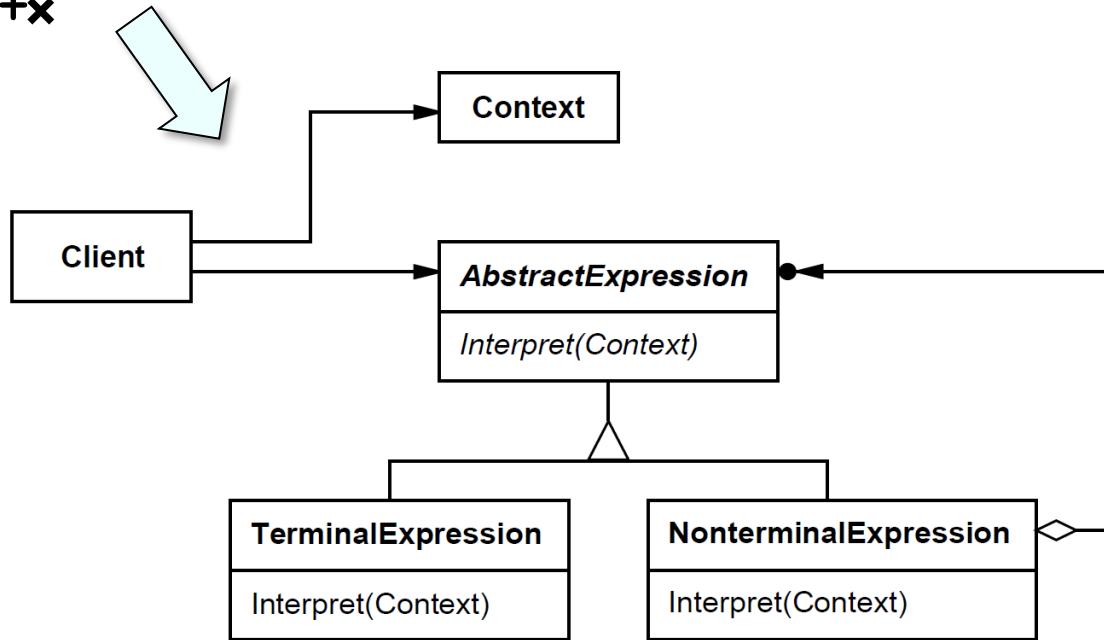


There are *many* classes in this design, but only a handful of patterns

Context: OO Expression Tree Processing App

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

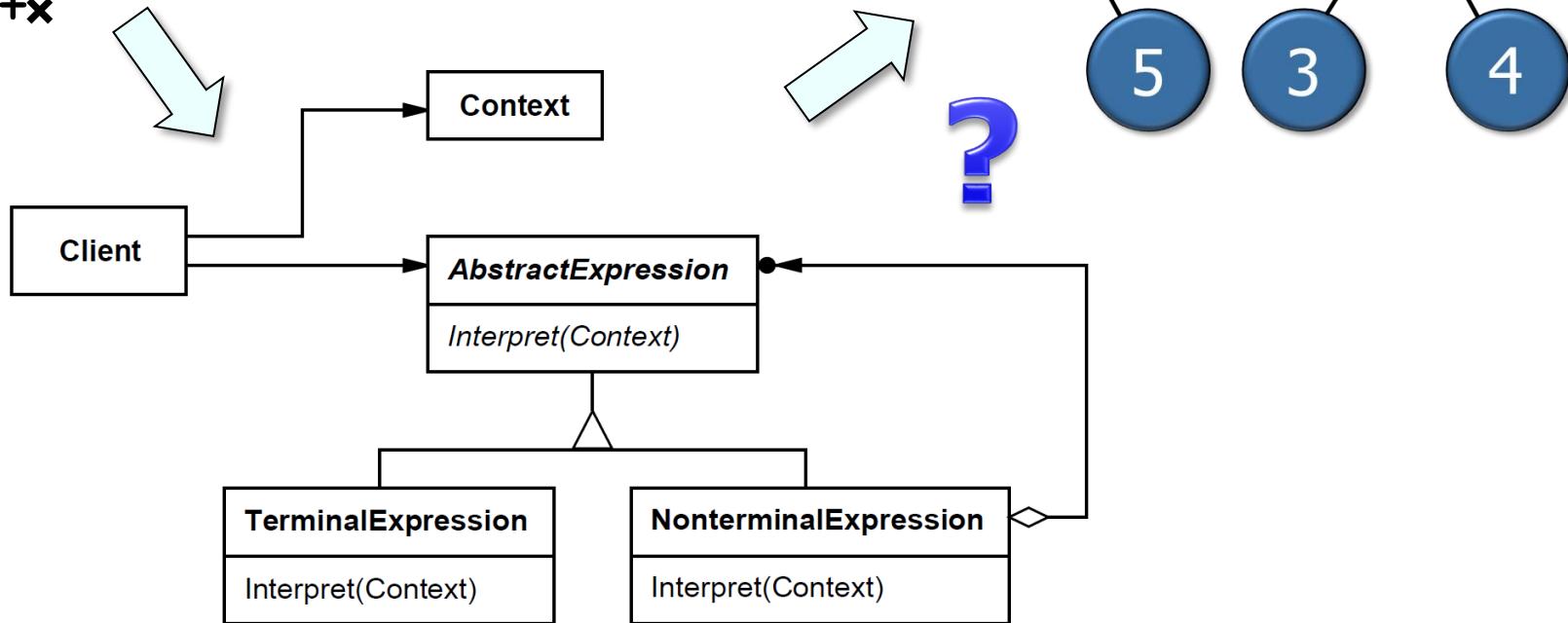
5~34+x



Context: OO Expression Tree Processing App

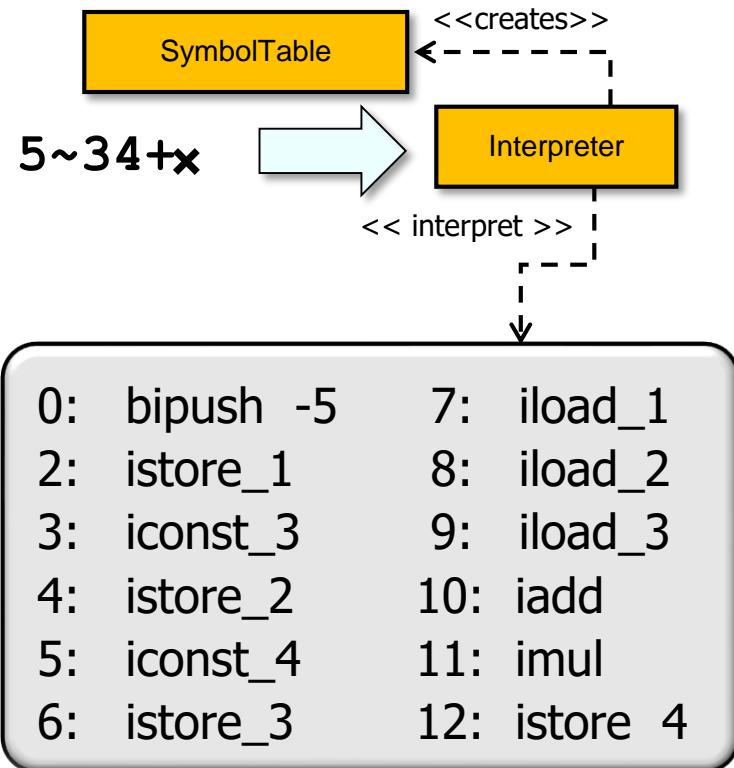
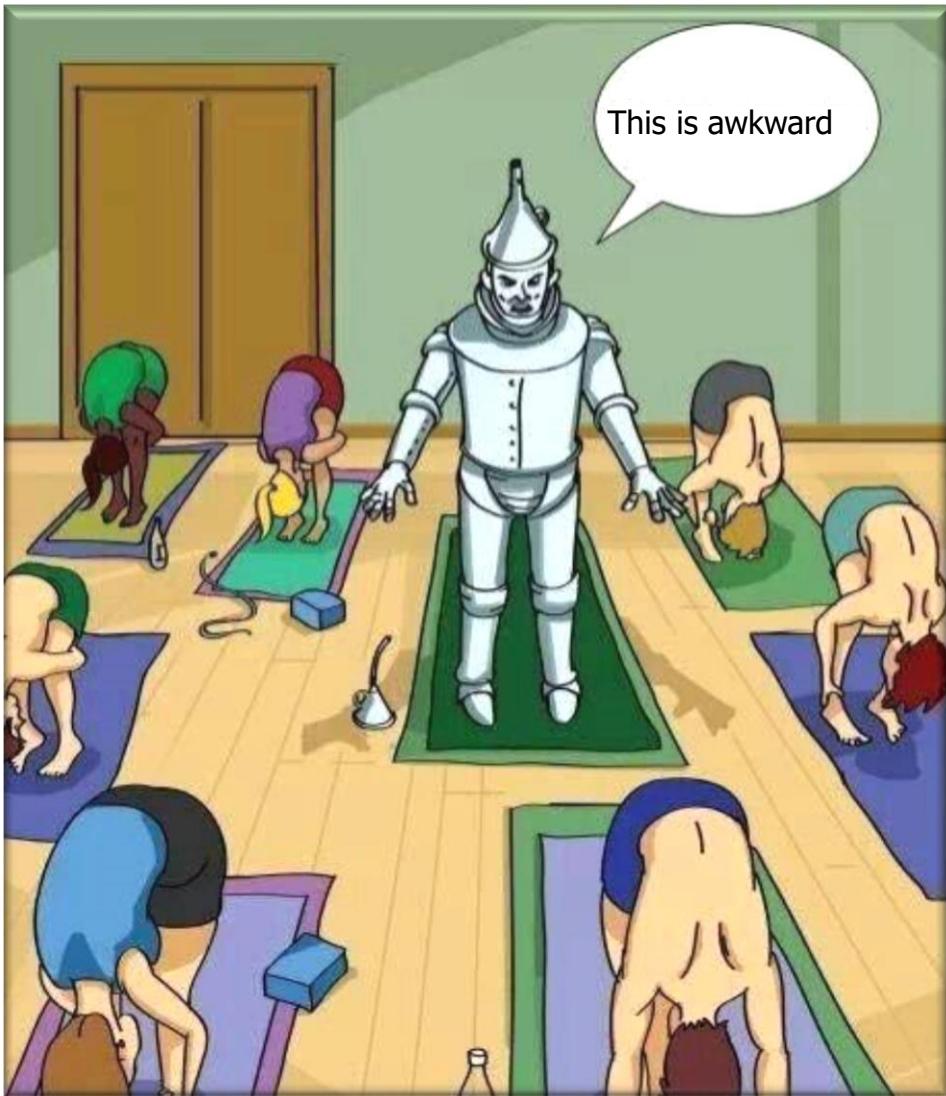
- Applying *Interpreter* helps to automate the parsing of user expression input
 - However, we still must determine how to convert this input into an expression tree

5~34+x



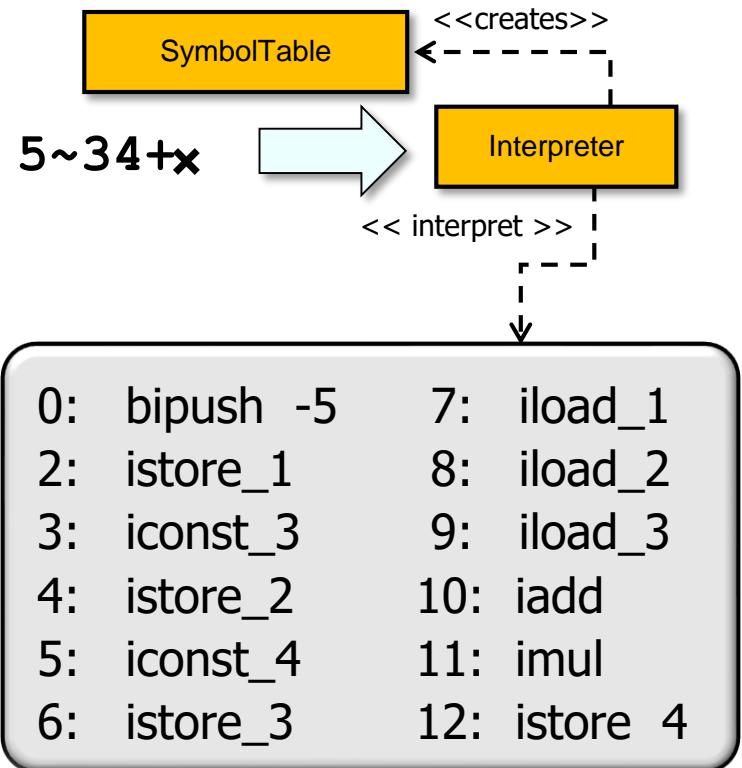
Problem: Inflexible Interpreter Output

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



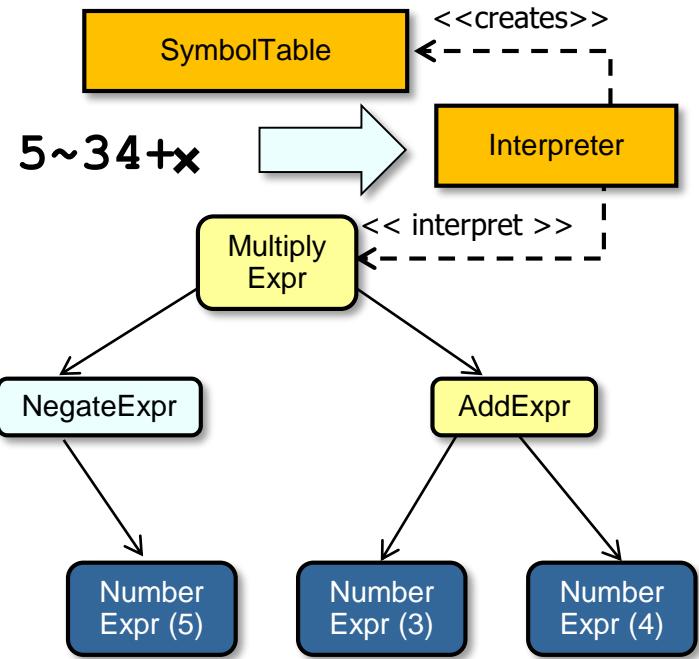
Problem: Inflexible Interpreter Output

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



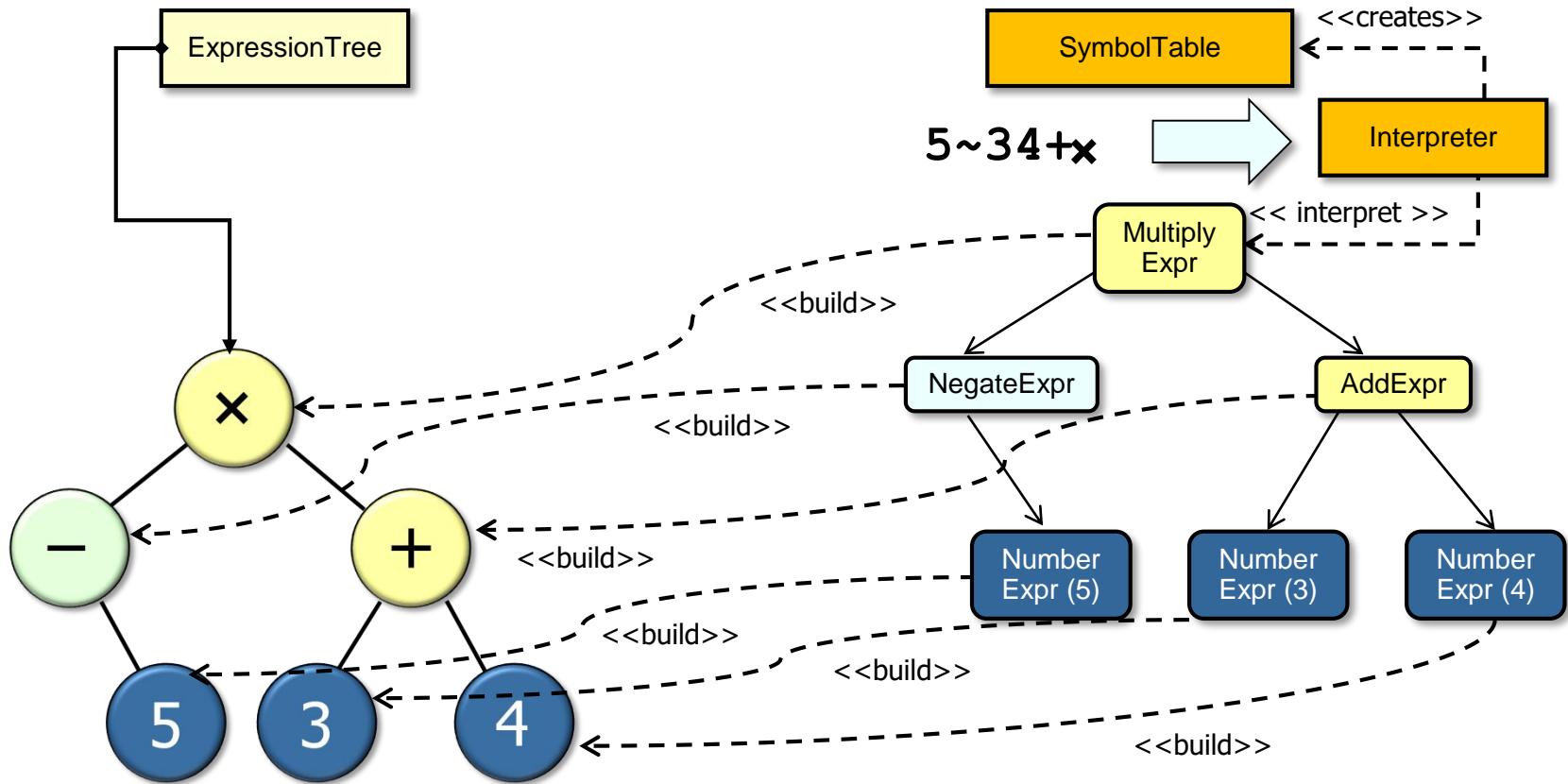
Solution: Build Complex Object Incrementally

- Have **Interpreter** create a parse tree
 - Optionally analyze & optimizes 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 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

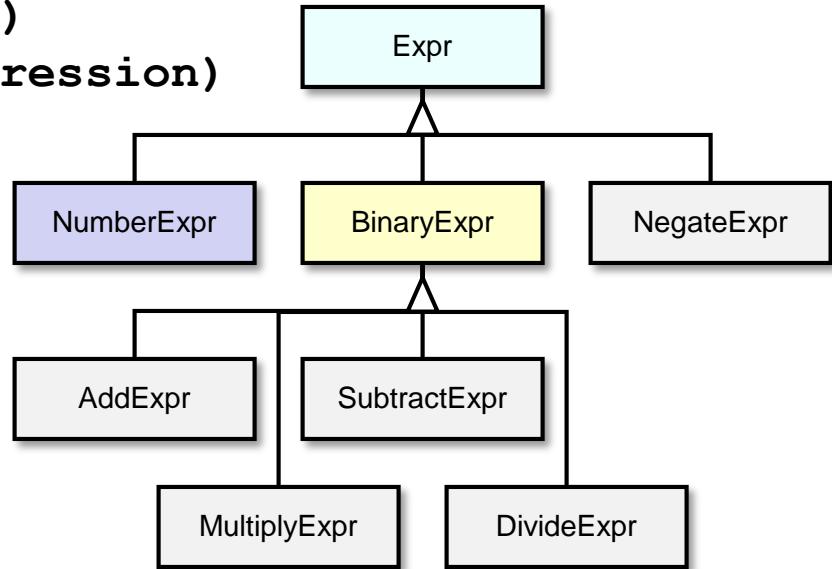
Class methods

void optimizeParseTree()

ExpressionTree buildExpressionTree()

ExpressionTree **interpret(String expression)**

Interpreter creates parse tree
incrementally by instantiating
Expr subclasses



Expr Class Hierarchy Overview

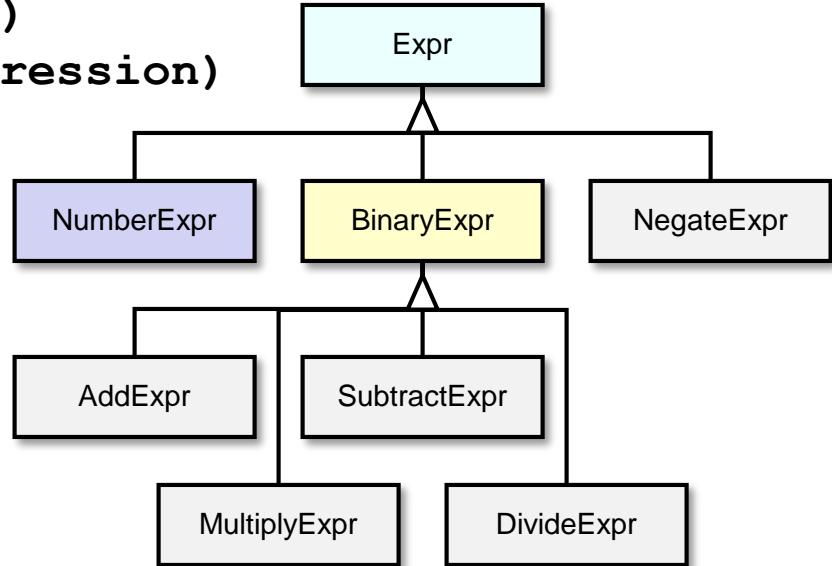
- Represents the nodes in the parse tree, which are used to build the corresponding nodes in the expression tree

Class methods

void [optimizeParseTree\(\)](#)

[ExpressionTree](#) [buildExpressionTree\(\)](#)

[ExpressionTree](#) [interpret\(String expression\)](#)



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

Elements of the Builder Pattern

Intent

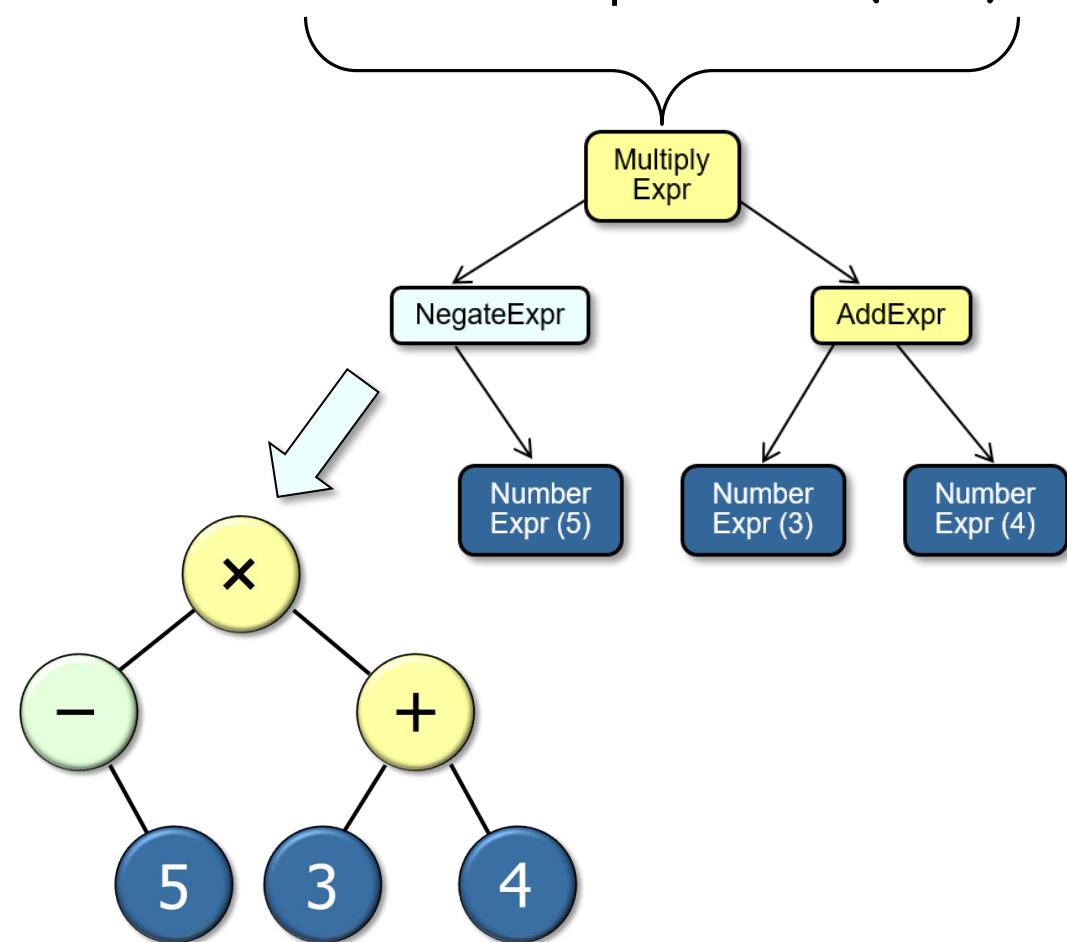
- Separate construction of a complex object from its representation

“pre-order” input = $x - 5 + 34$

“post-order” input = $5 - 34 + x$

“level-order” input = $x - + 5 3 4$

“in-order” input = $- 5 x (3 + 4)$

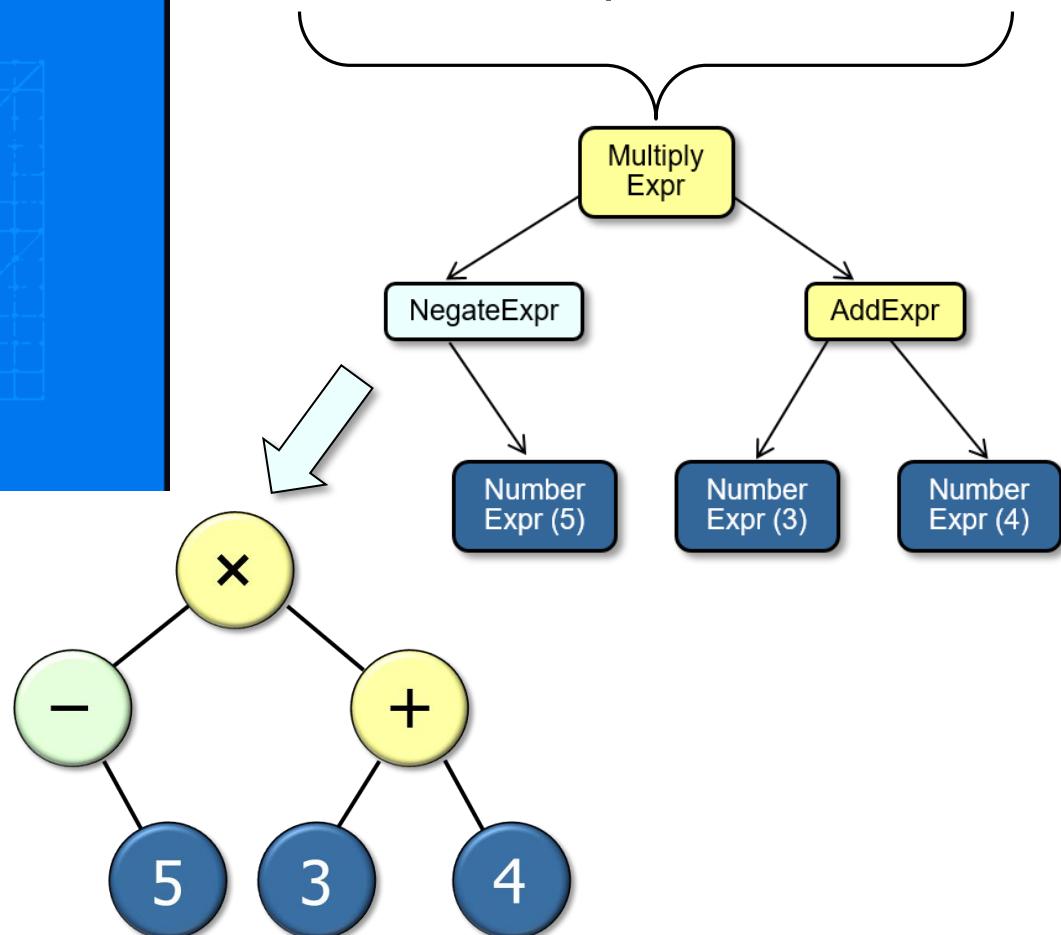


Applicability

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



“pre-order” input = $x - 5 + 34$
“post-order” input = $5 - 34 + x$
“level-order” input = $x - + 5 3 4$
“in-order” input = $- 5 x (3 + 4)$

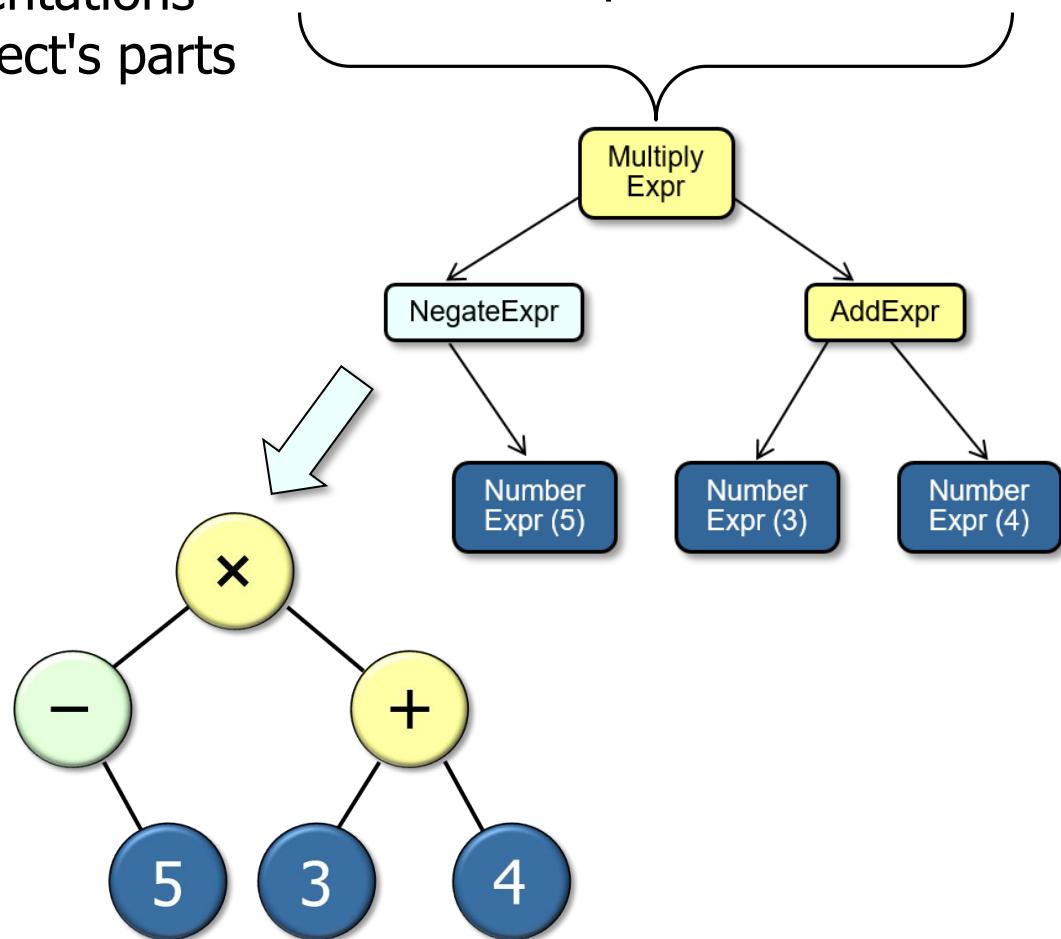


Applicability

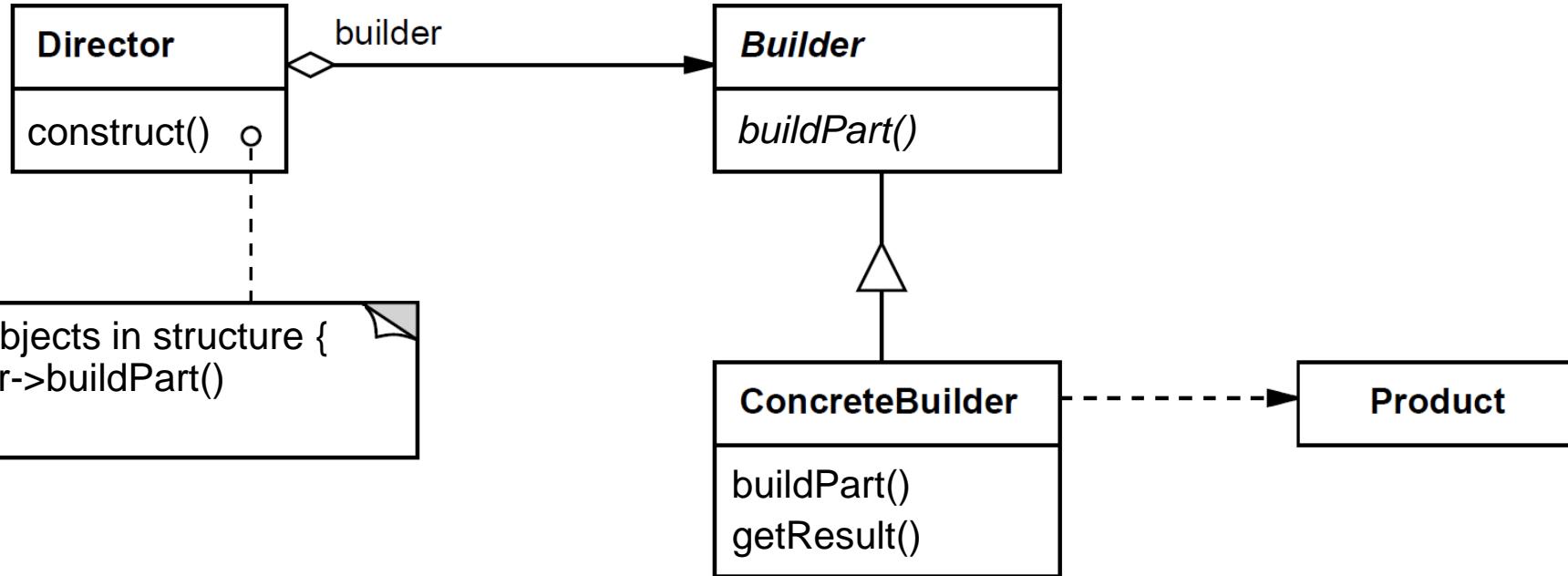
- Need to isolate knowledge of creating a complex object from its parts
- Need to allow different implementations & (internal) interfaces of an object's parts



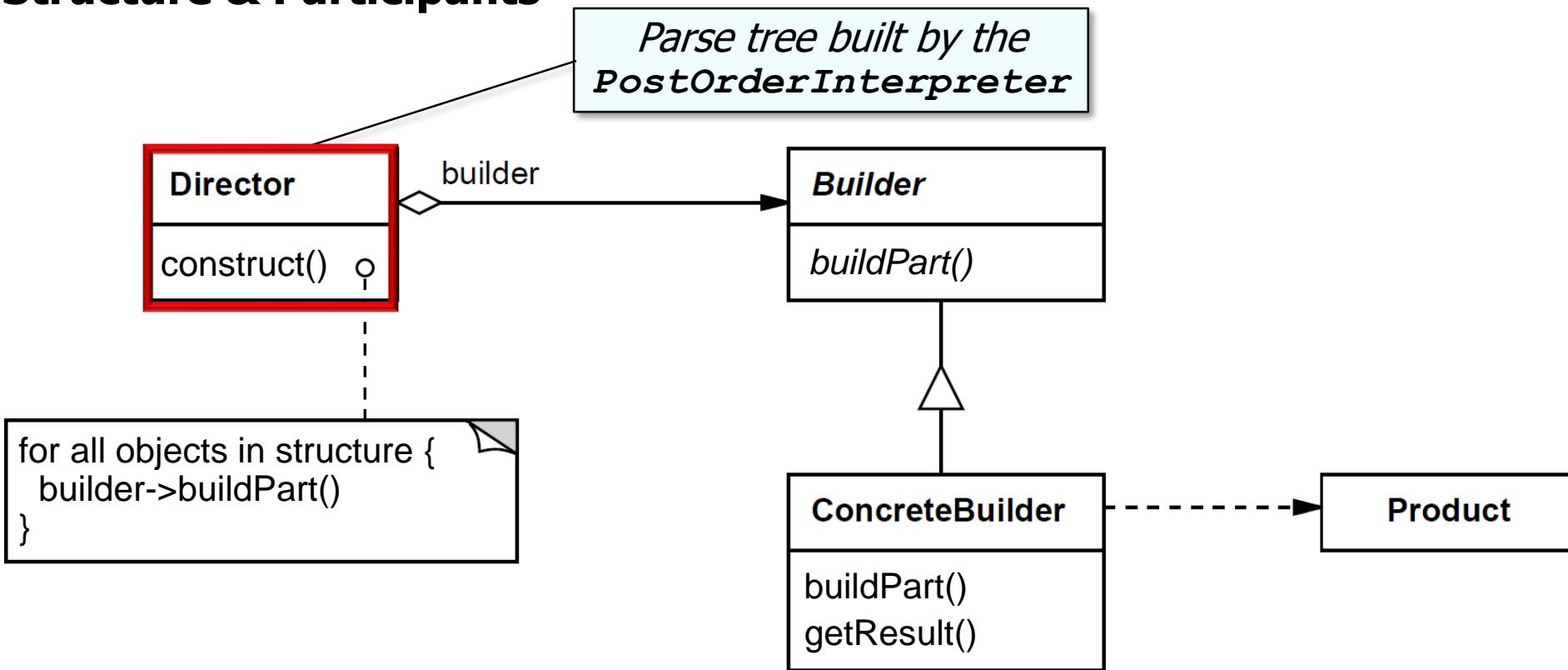
“pre-order” input = $x - 5 + 34$
“post-order” input = $5 - 34 + x$
“level-order” input = $x - + 5 3 4$
“in-order” input = $- 5 x (3 + 4)$



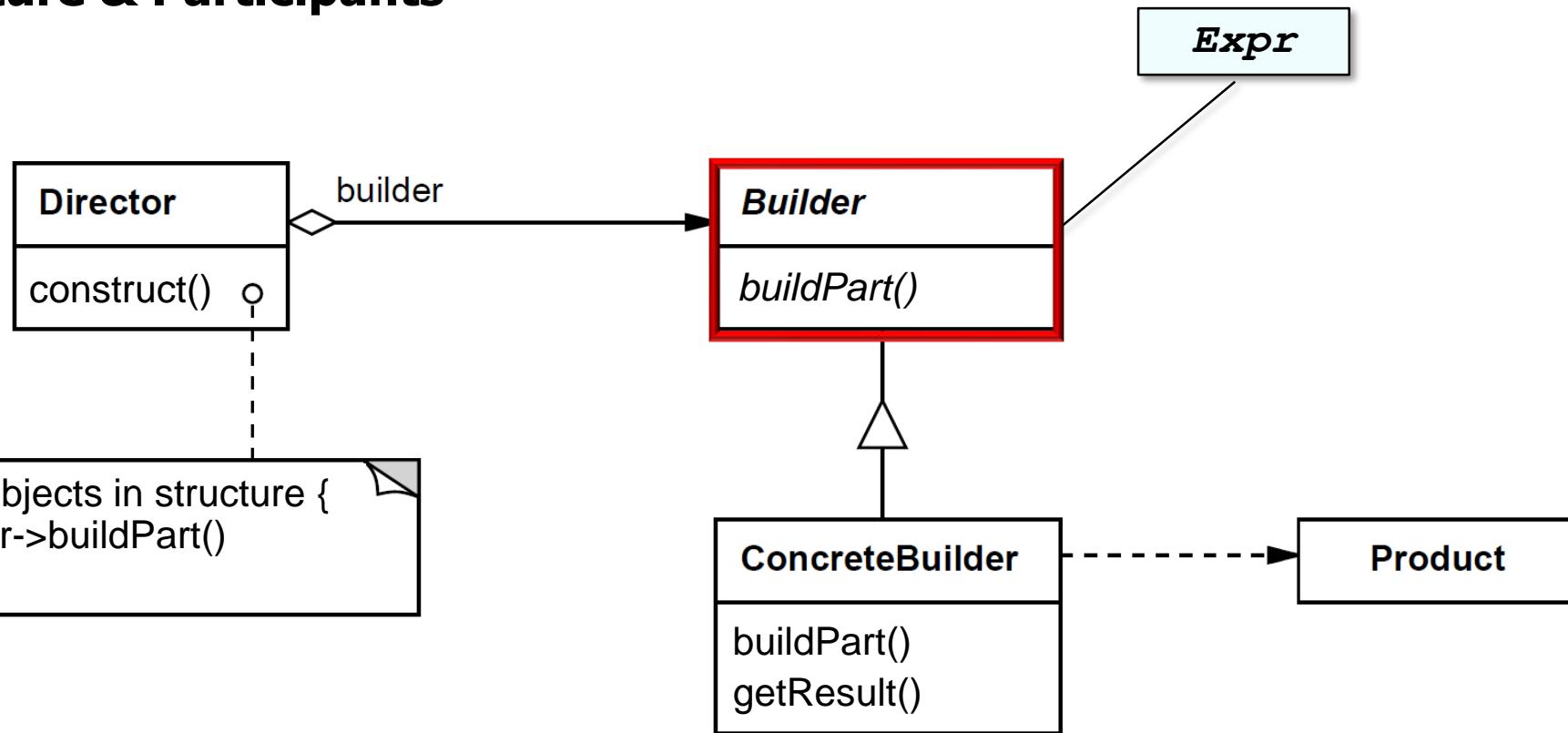
Structure & Participants



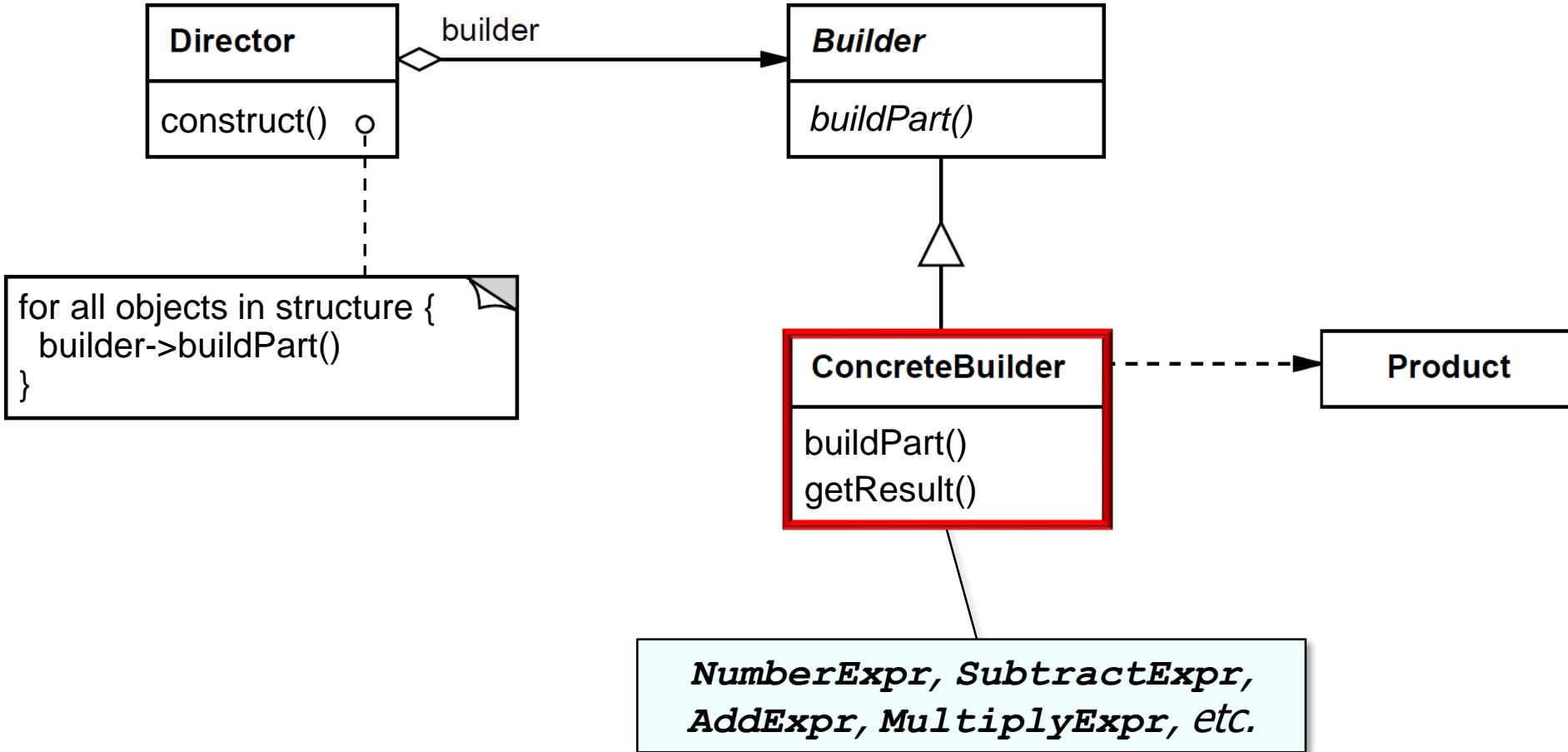
Structure & Participants



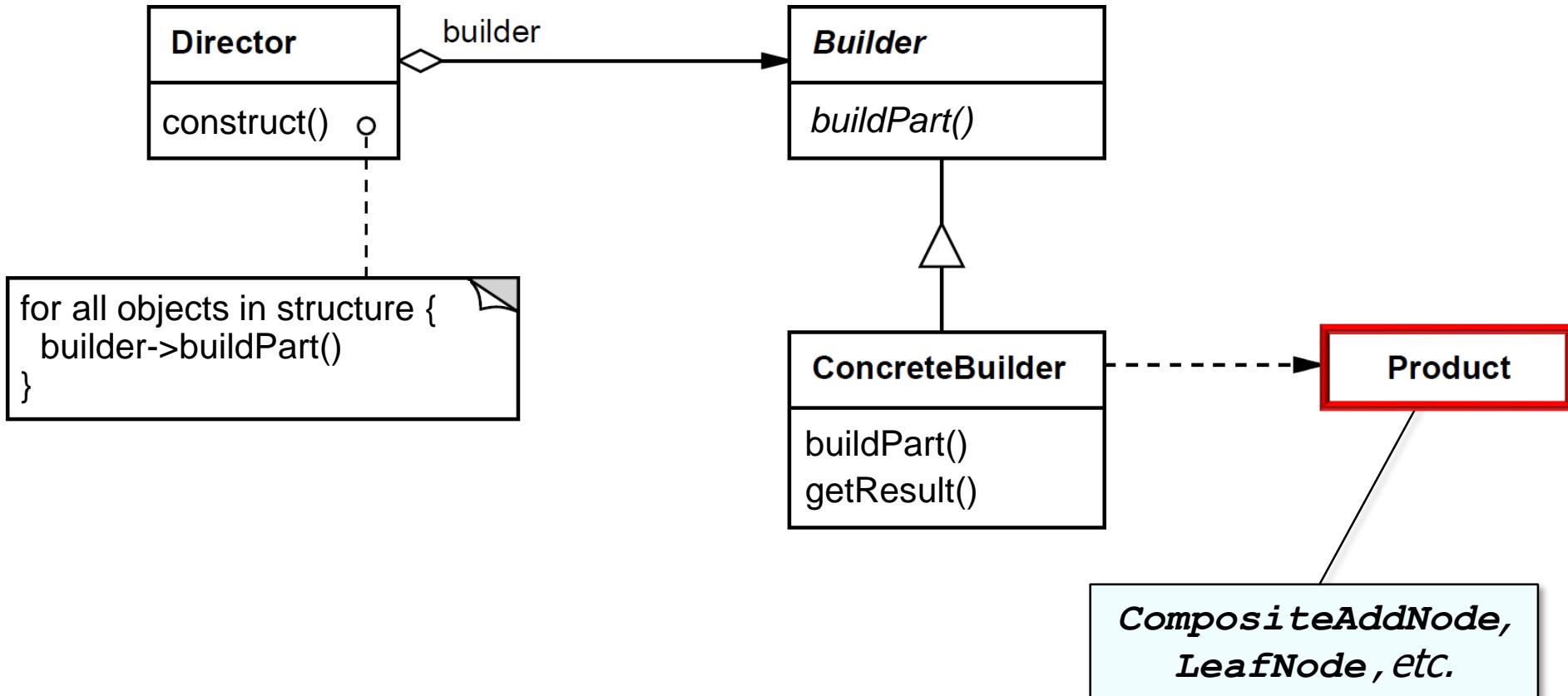
Structure & Participants



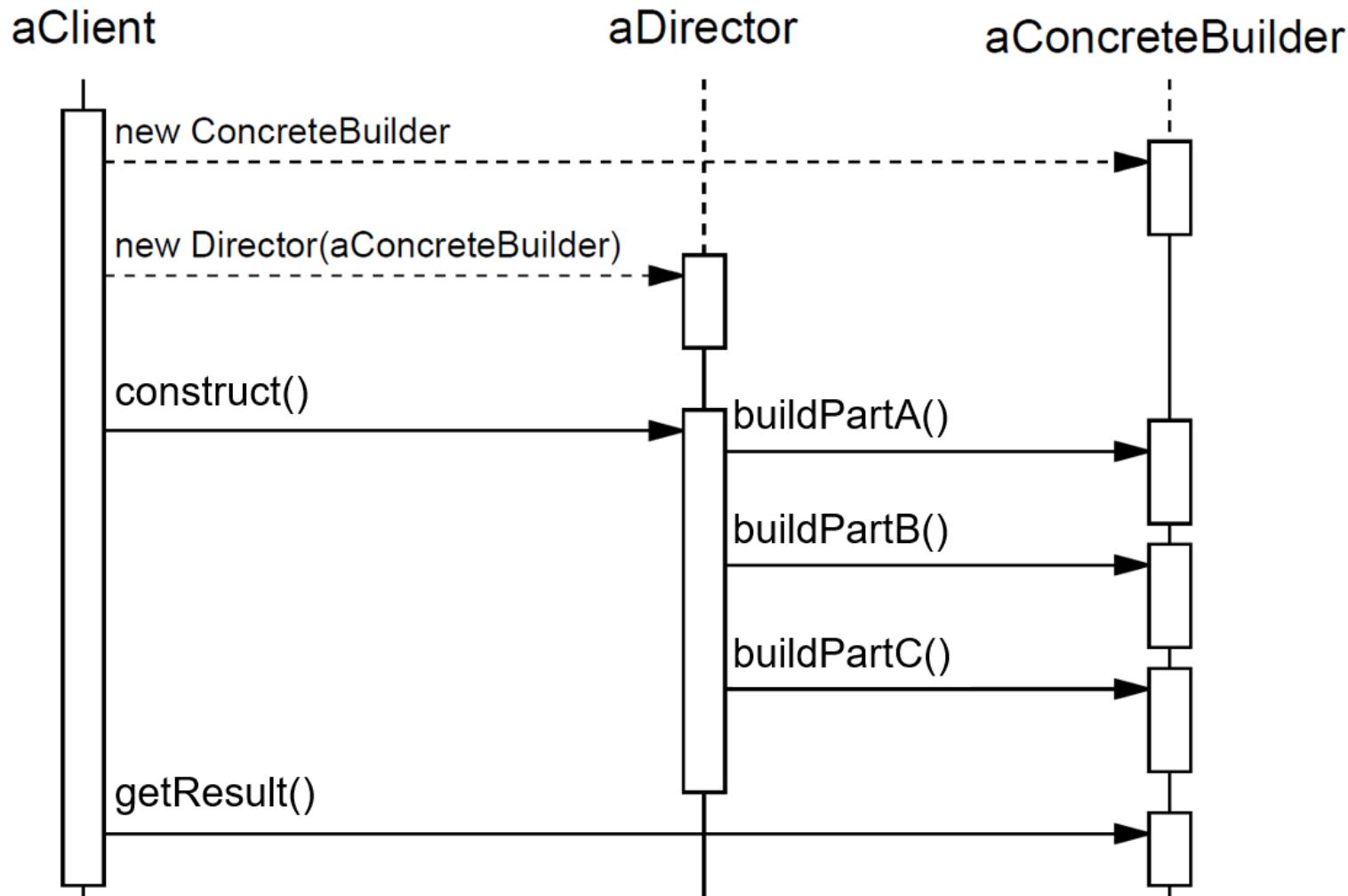
Structure & Participants



Structure & Participants

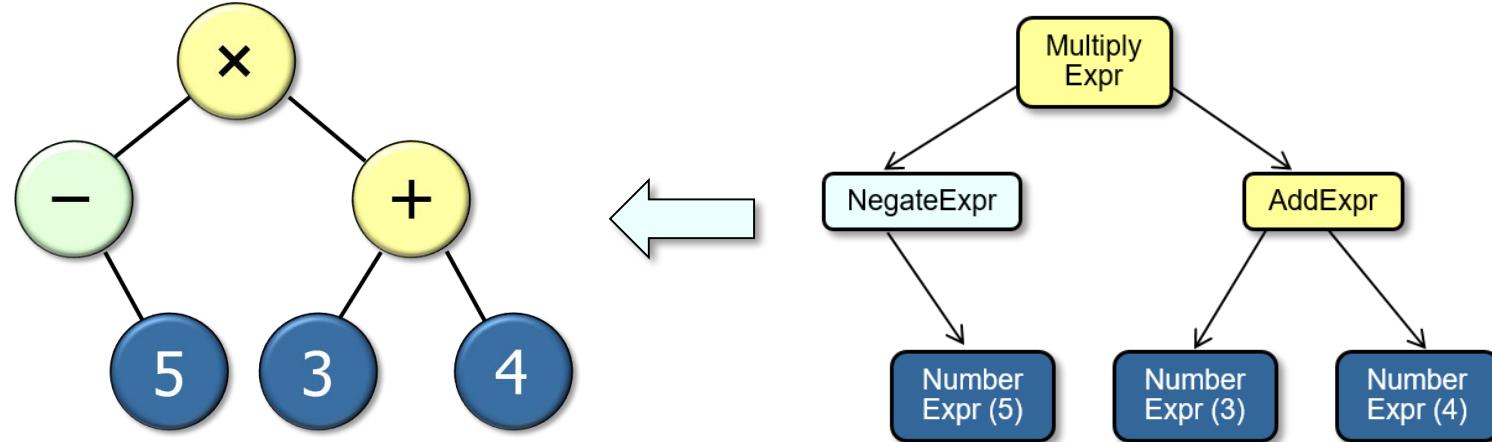


Collaborations



Builder example in Java

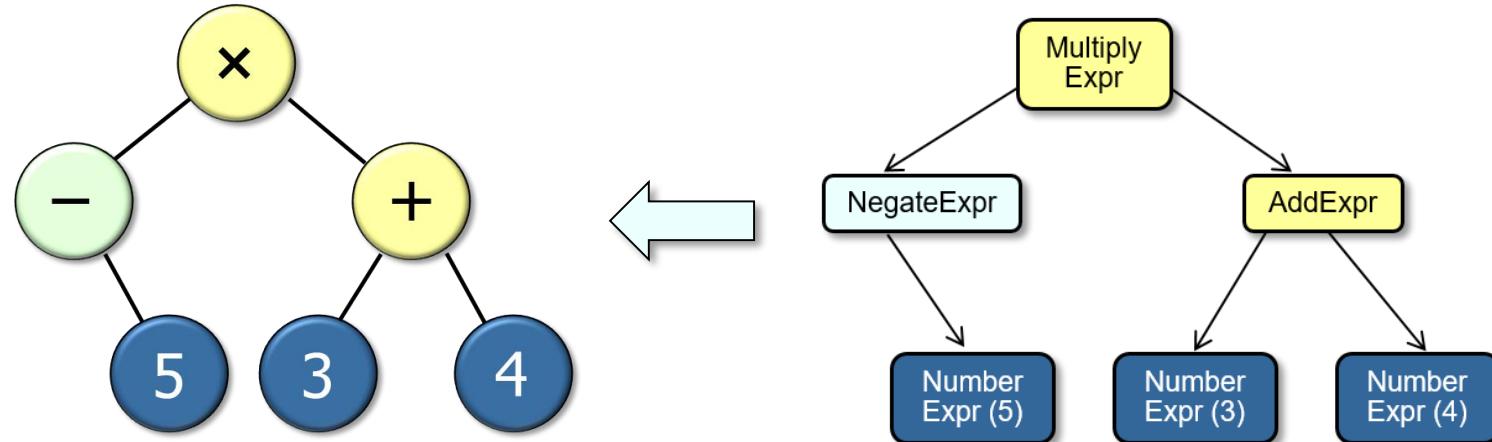
- `buildExpressionTree()` builds composite expression tree from parse tree



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

Builder example in Java

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



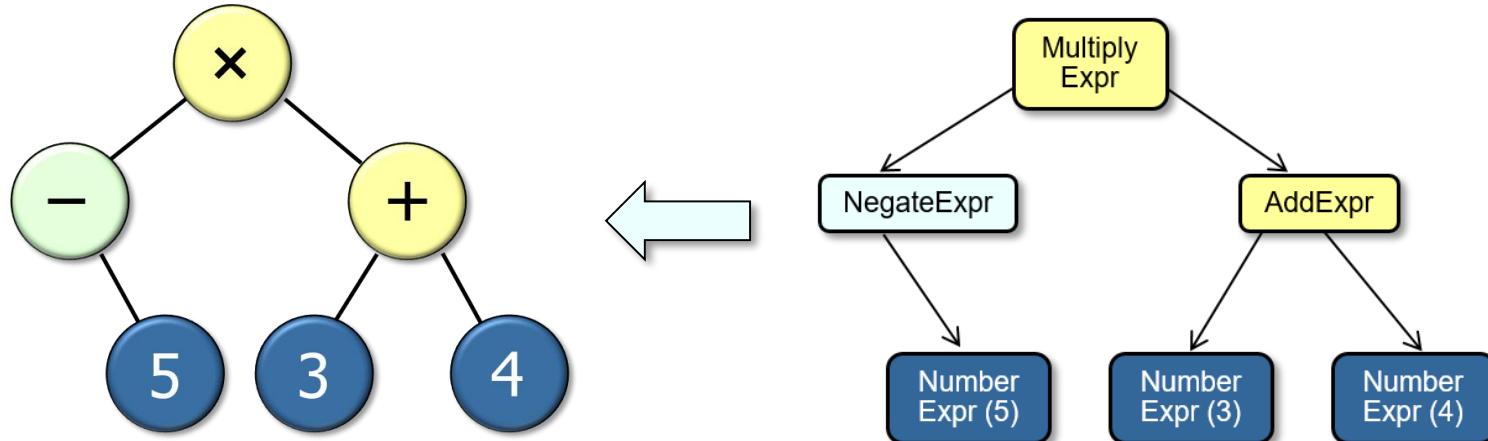
```
public class PostOrderInterpreter extends InterpreterImpl {  
    protected ExpressionTree buildExpressionTree(Expr parseTree) {  
        return expressionTreeFactory  
            .makeExpressionTree(parseTree.build());  
    }  
}
```

Invoke a recursive expression tree build, starting w/root expr in parse tree created by PostOrderInterpreter



Builder example in Java

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



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

Build component nodes recursively



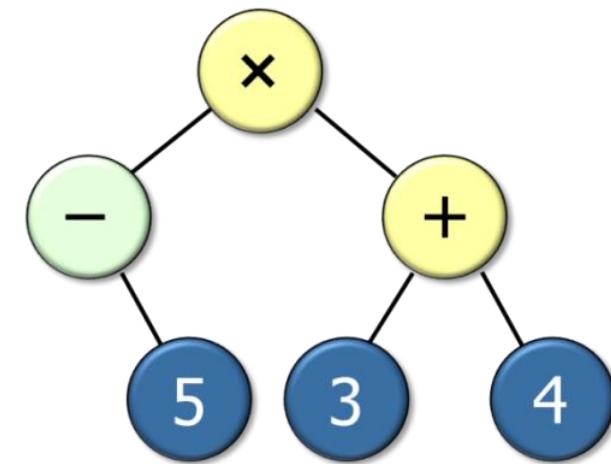
```
class NumberExpr  
    extends Expr {  
ComponentNode build() {  
    return new LeafNode(mItem);  
}  
...  
}
```

Consequences

- + Isolates code for construction & representation

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

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



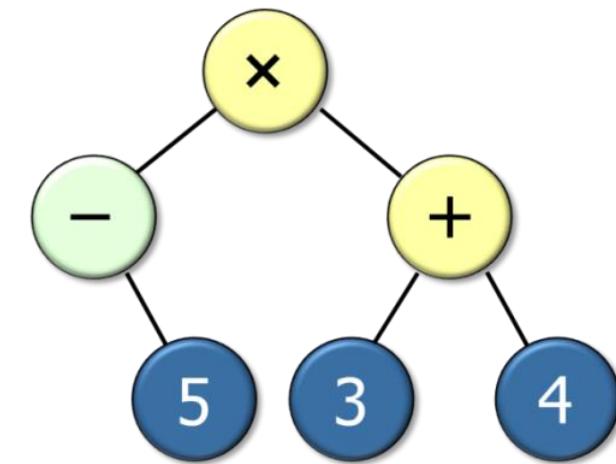
Consequences

- + Isolates code for construction & representation
- + Finer control over the construction process

Every composite node controls how it's constructed

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

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

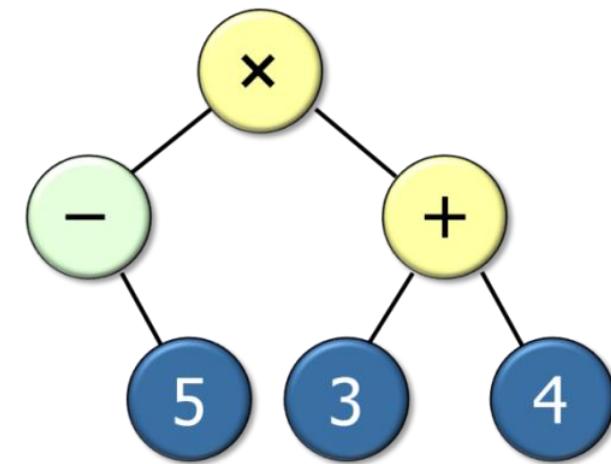


Consequences

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

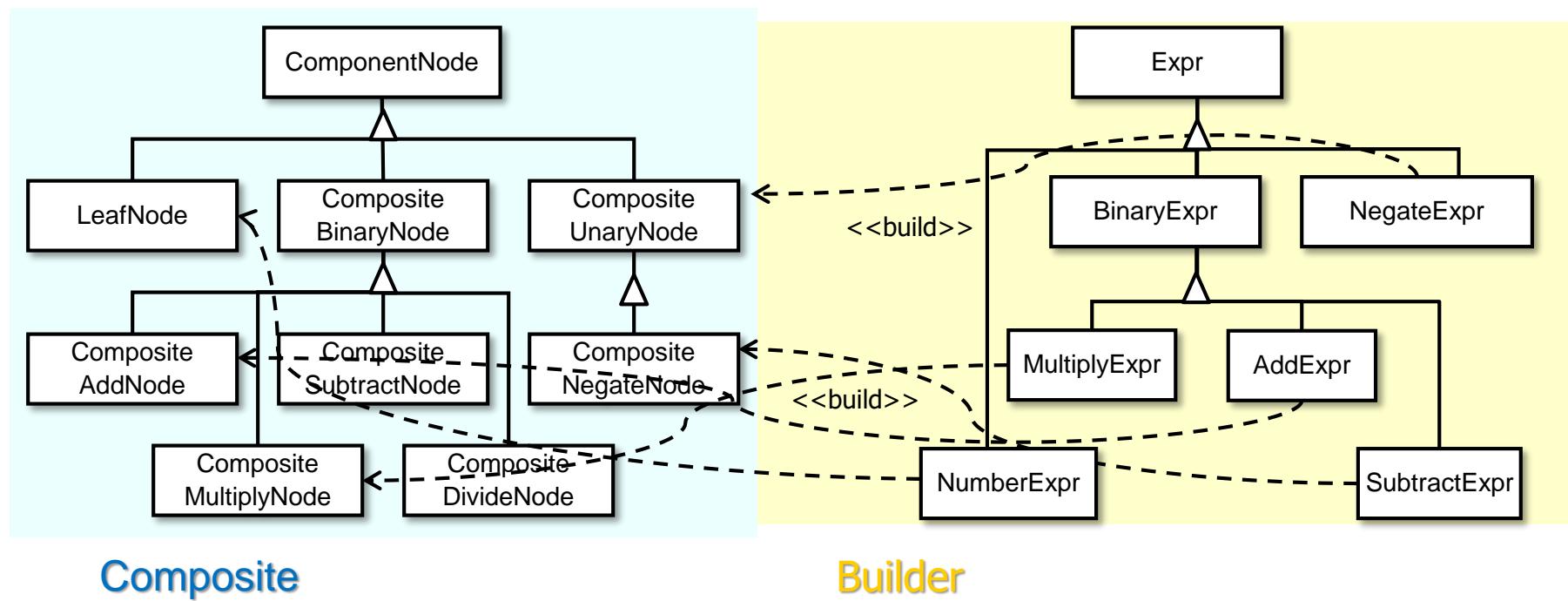
```
class AddExpr extends BinaryExpr {  
    ComponentNode build() {  
        return new TreeNode('+',  
            (left.build(), right.build());  
    }  
}
```

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



Consequences

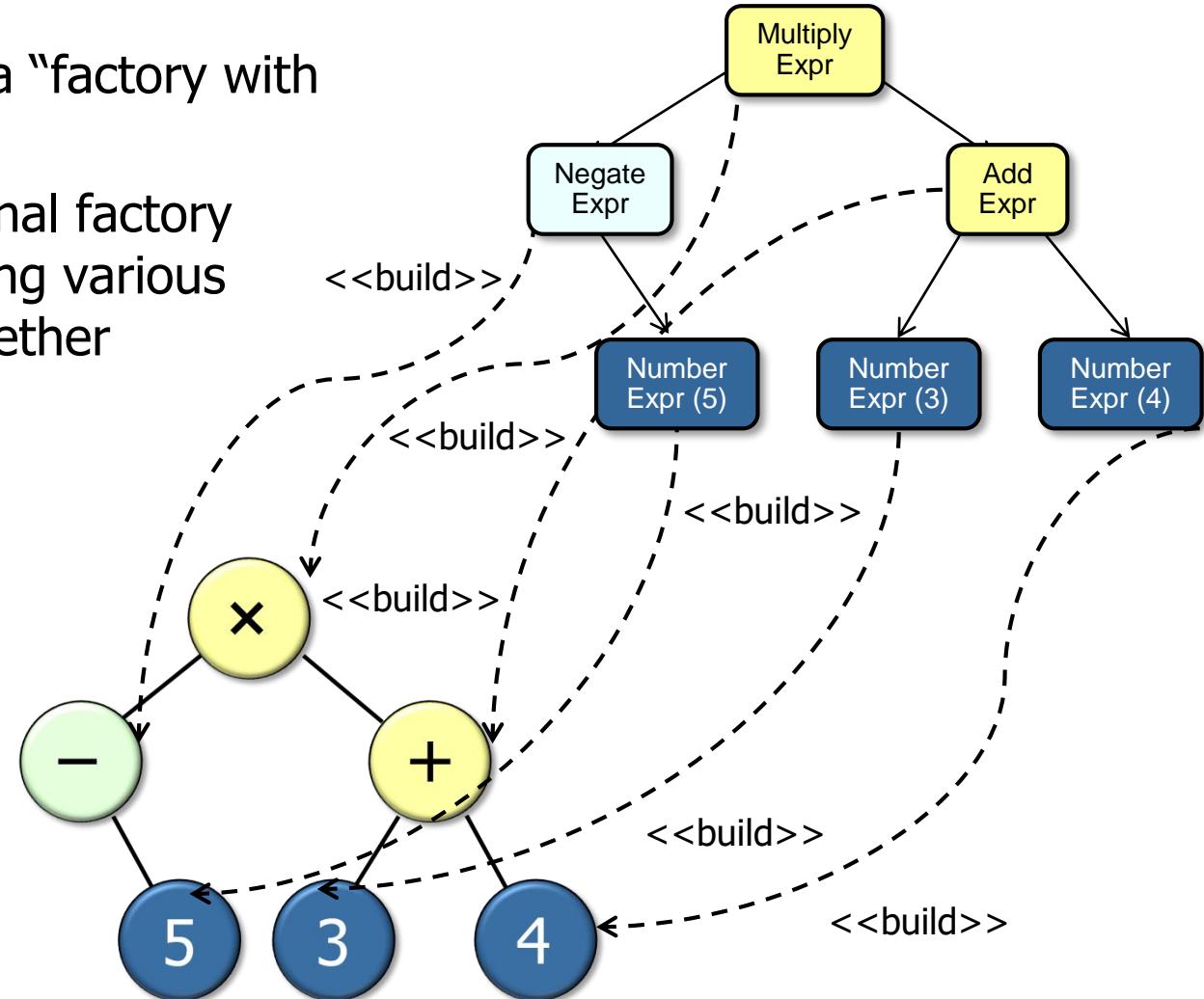
- May involve a lot of classes & class interdependencies



Knowledge of patterns helps to reduce “surface area” of these many classes

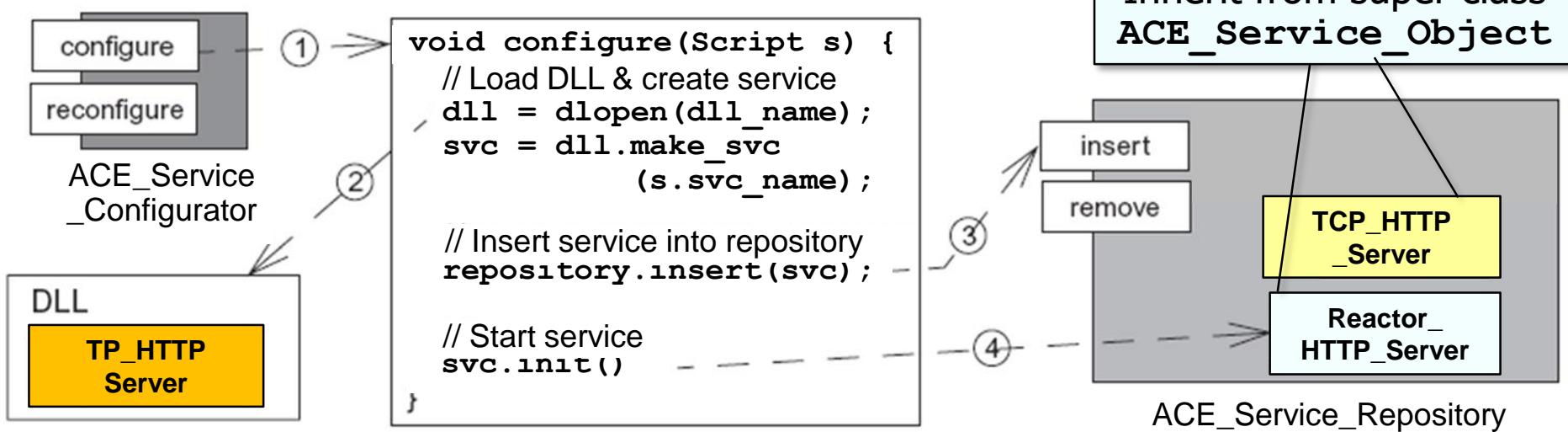
Implementation

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



Known Uses

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



Known Uses

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

Builds a new object

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

Known Uses

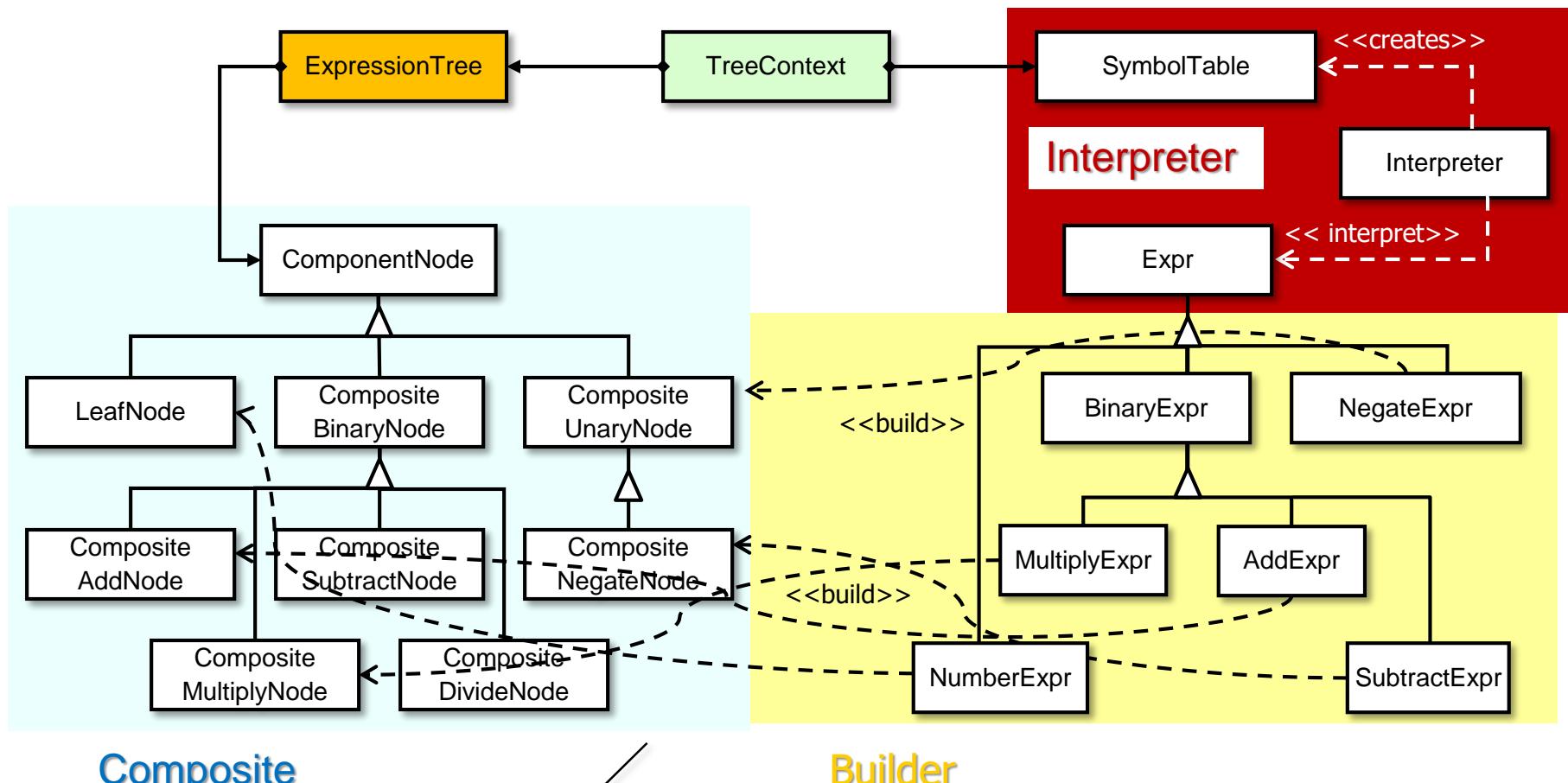
- ET++ RTF converter
- Smalltalk-80
- ACE Service Configurator framework
- “Effective Java” style

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

```
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); }  
    ...  
}  
...  
Note the use of "fluent interface" API design style
```

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



Composite

Builder

Interpreter, Builder, & Composite are a useful "pattern sequence"

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

End of the
Builder Pattern

The Command Pattern

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software
Integrated Systems

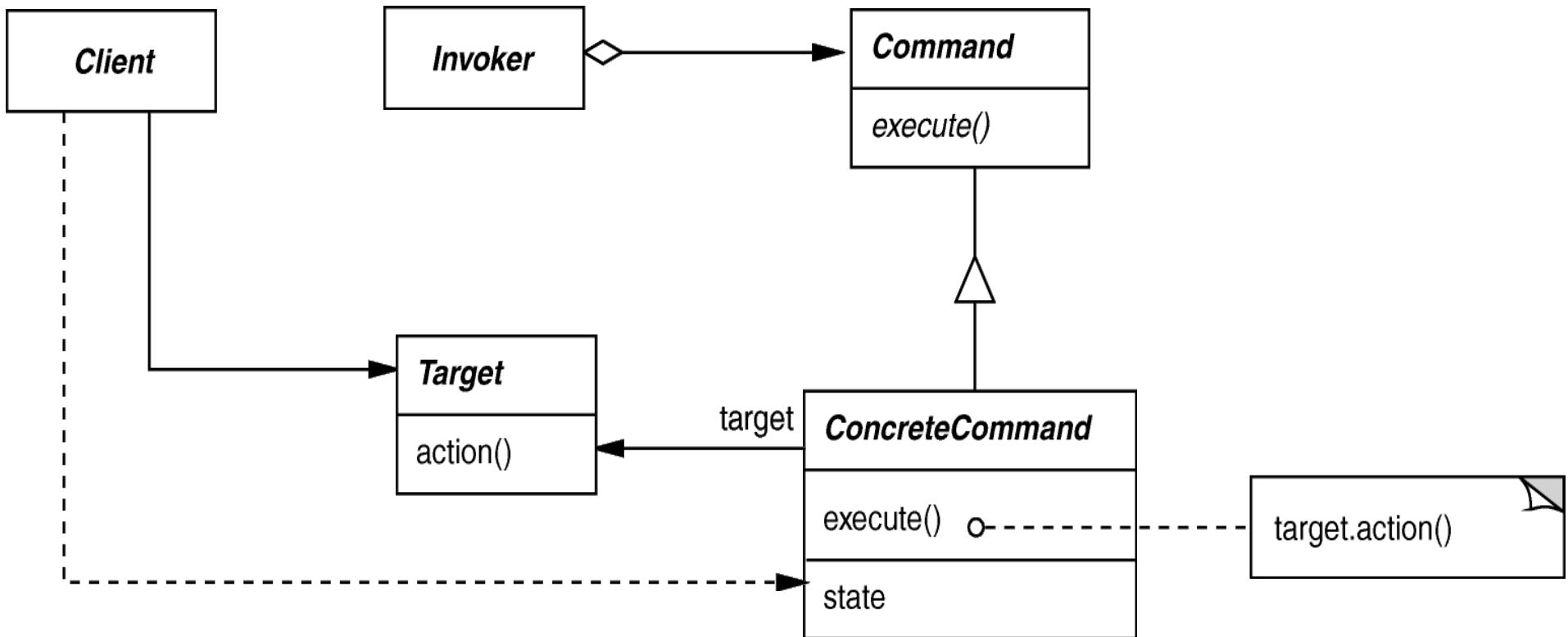
Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives

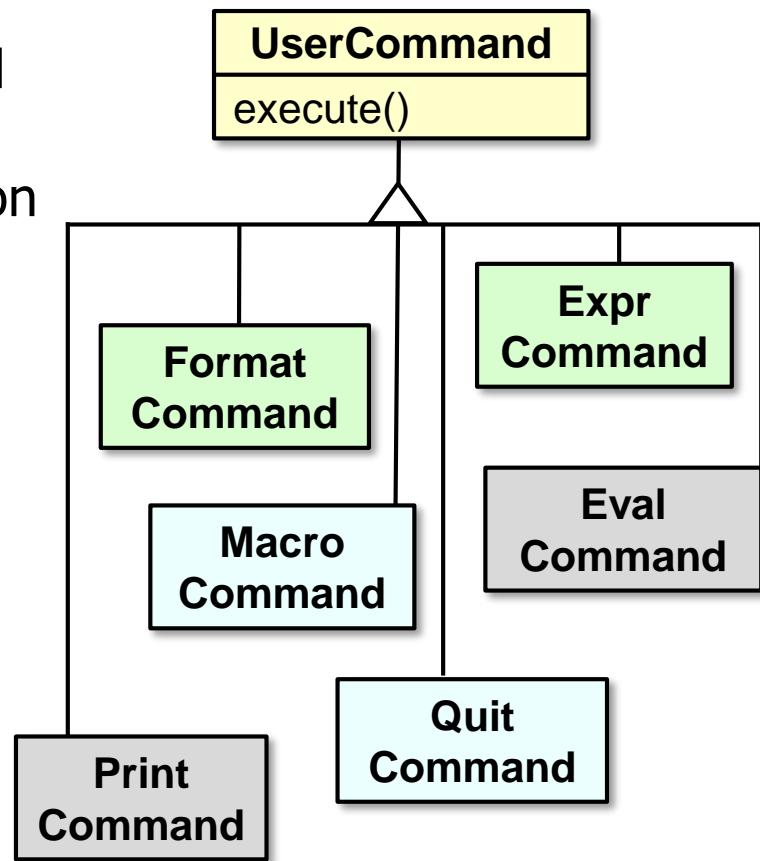
Learning Objectives

- Understand the *Command* pattern



Learning Objectives

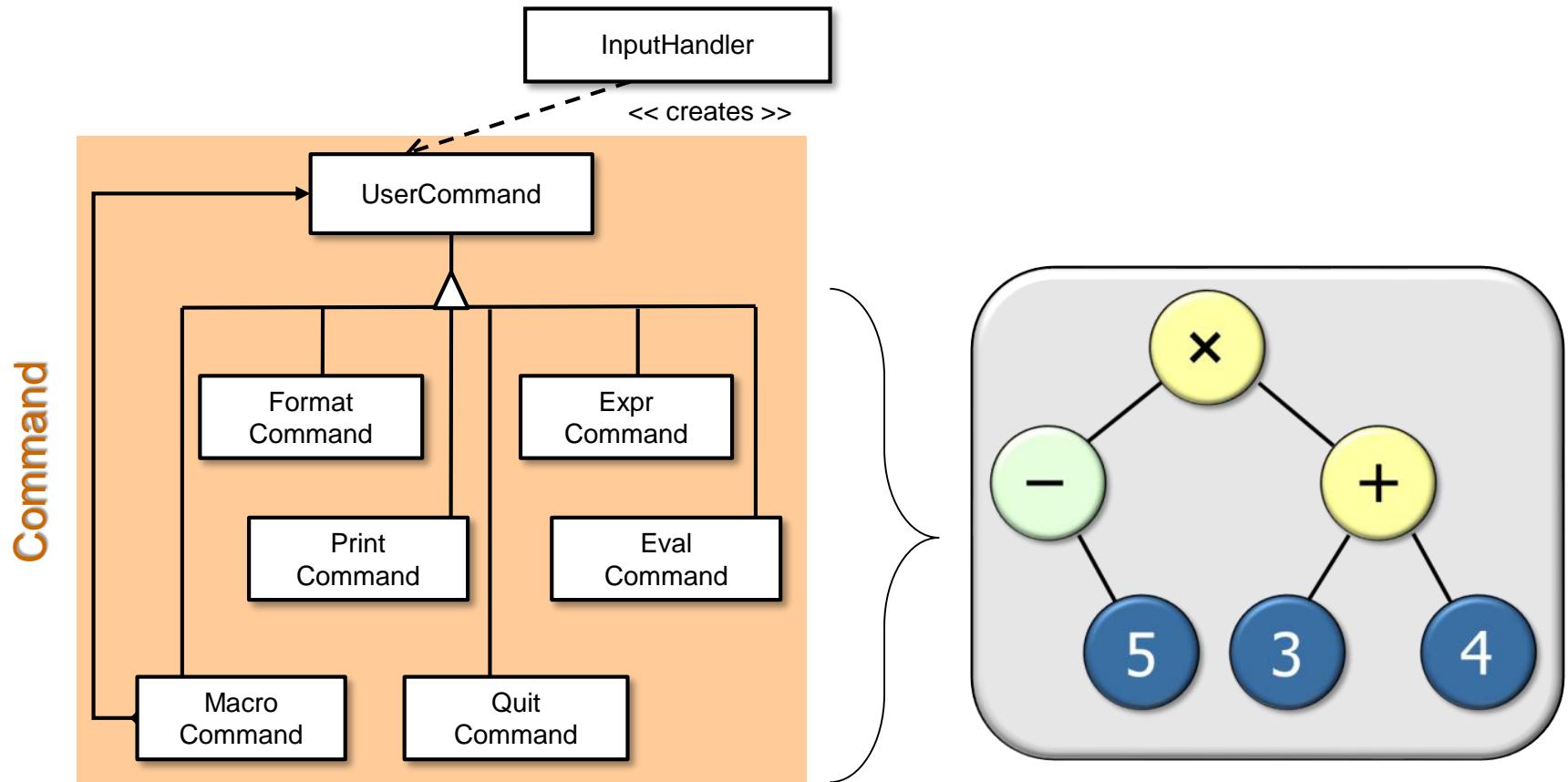
- Understand the *Command* pattern
- Recognize how *Command* can be applied to perform user-requested commands consistently & extensibly in the expression tree processing app



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 & extensibly in the expression tree processing app



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

Context: OO Expression Tree Processing App

- Supports execution of user commands in verbose mode

The screenshot shows a Windows-style console window titled "Console". The title bar also displays "Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Jul 15, 2013 12:51:24 PM)". The window contains the following text:

```
Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Jul 15, 2013 12:51:24 PM)

1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order| level-order]
0. quit

> format in-order

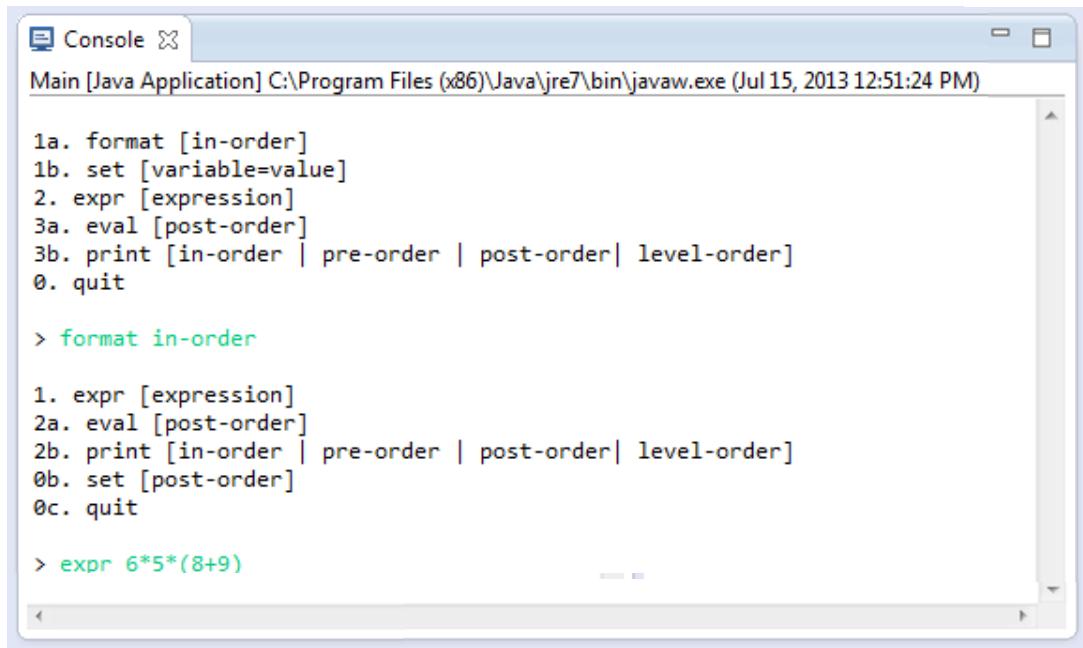
1. expr [expression]
2a. eval [post-order]
2b. print [in-order | pre-order | post-order| level-order]
0b. set [post-order]
0c. quit

> expr 6*5*(8+9)
```

Verbose mode

Context: OO Expression Tree Processing App

- Support macro commands in succinct mode



```
Console X
Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (Jul 15, 2013 12:51:24 PM)

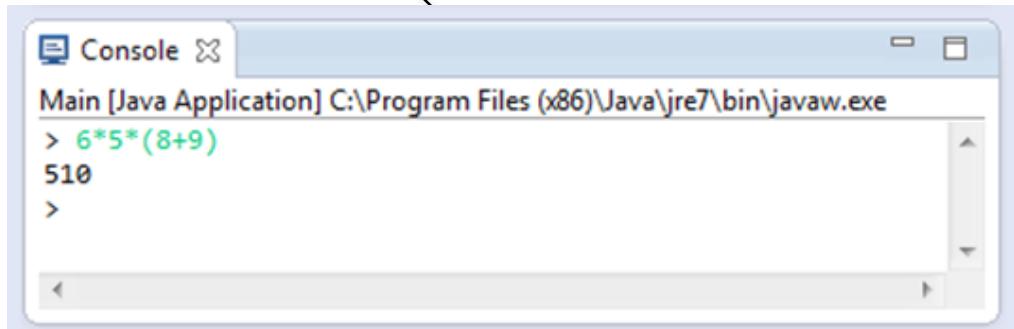
1a. format [in-order]
1b. set [variable=value]
2. expr [expression]
3a. eval [post-order]
3b. print [in-order | pre-order | post-order| level-order]
0. quit

> format in-order

1. expr [expression]
2a. eval [post-order]
2b. print [in-order | pre-order | post-order| level-order]
0b. set [post-order]
0c. quit

> expr 6*5*(8+9)
```

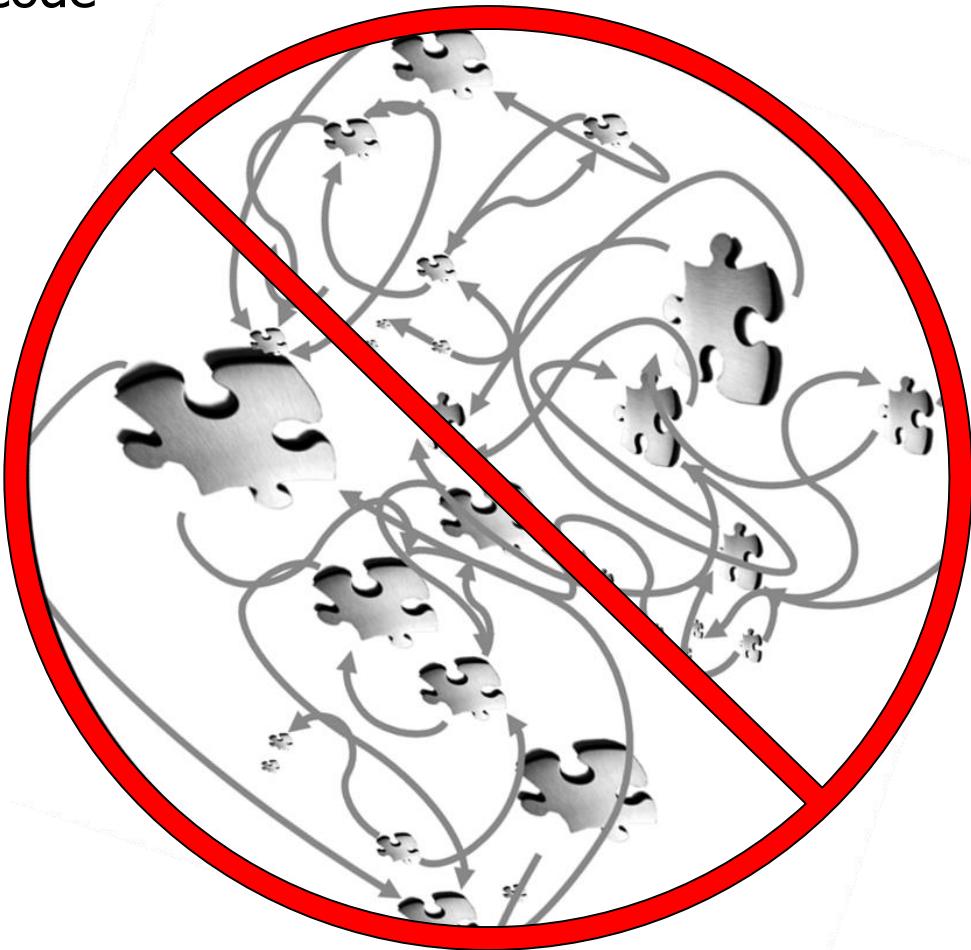
Succinct mode



```
Console X
Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe
> 6*5*(8+9)
510
>
```

Problem: Scattered/Fixed User Request Implementations

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



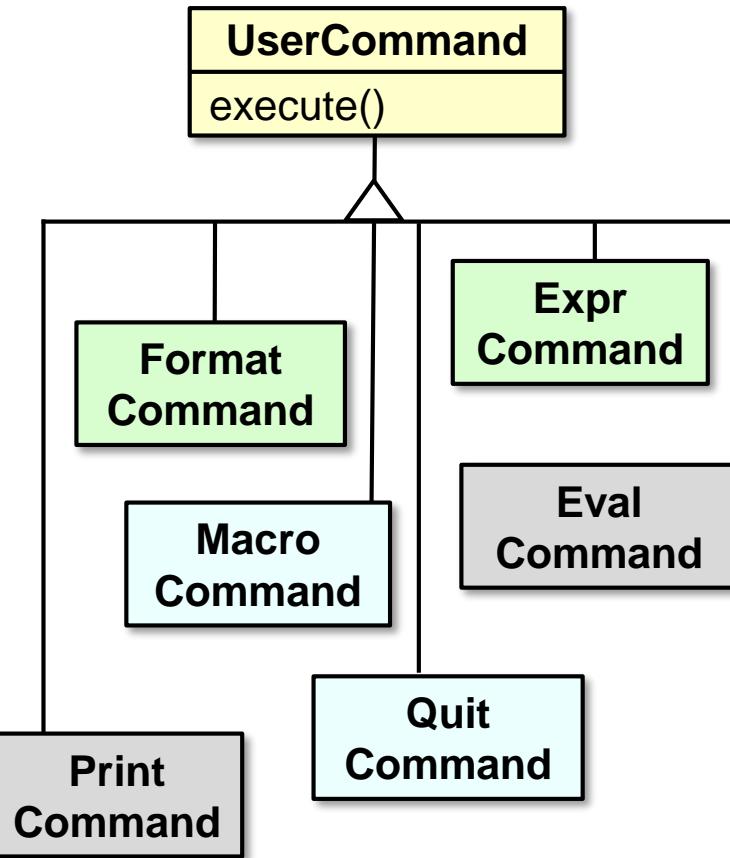
Problem: Scattered/Fixed User Request Implementations

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



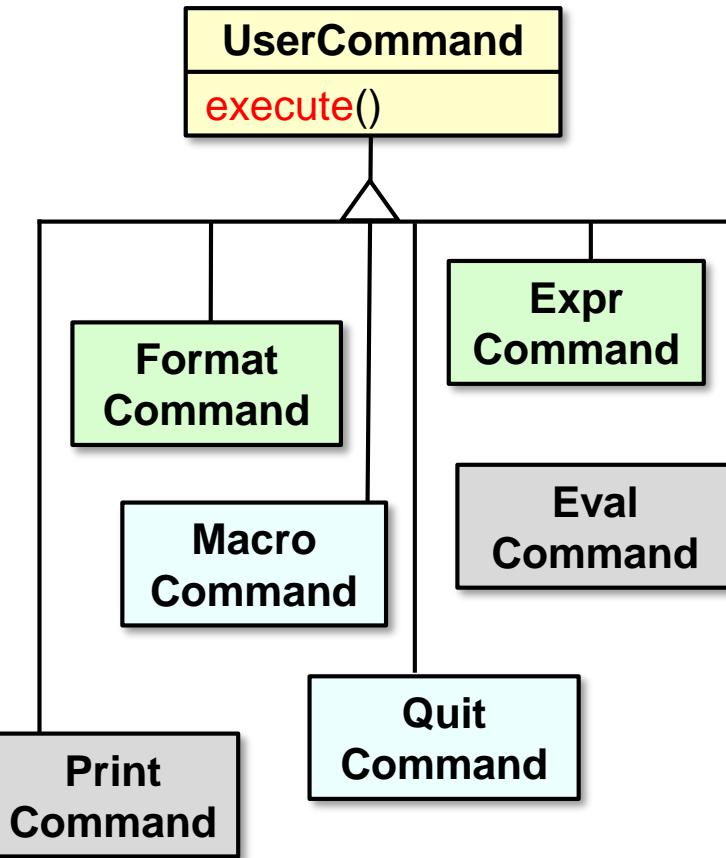
Solution: Encapsulate User Requests as Commands

- Create a hierarchy of `UserCommand` subclasses



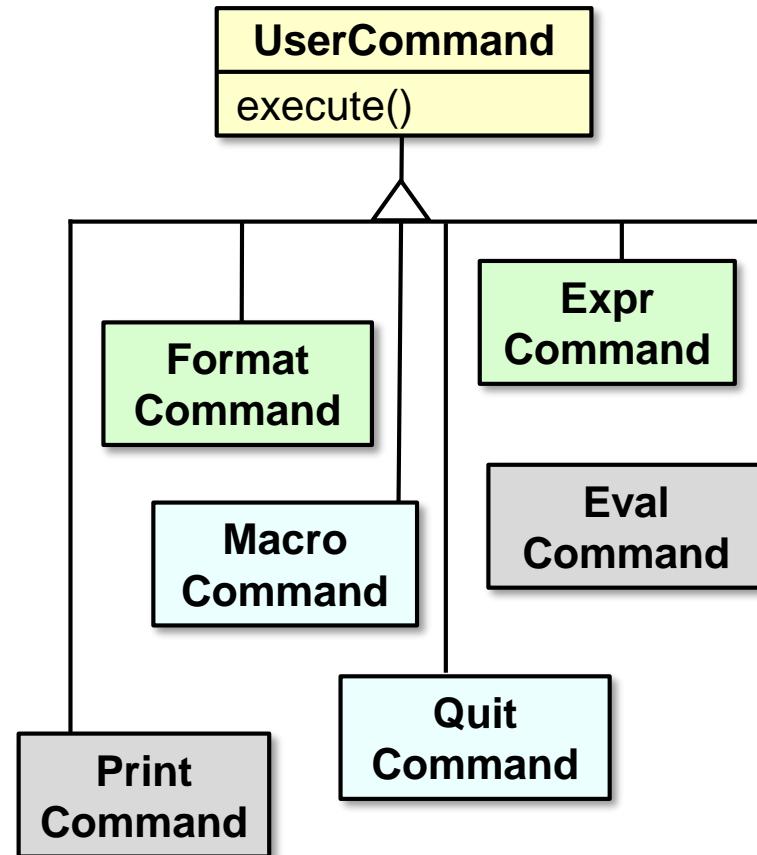
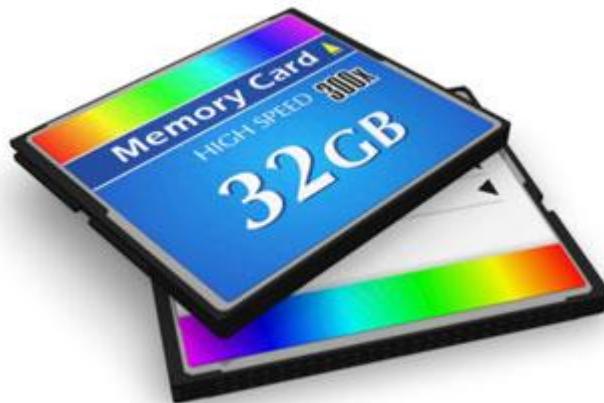
Solution: Encapsulate User Requests as Commands

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



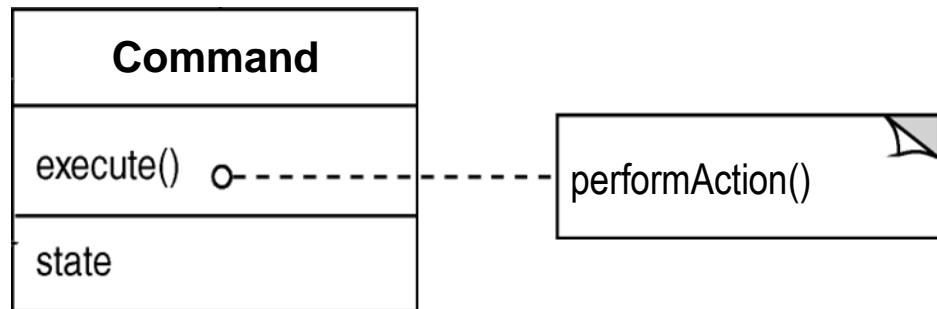
Solution: Encapsulate User Requests as Commands

- Create a hierarchy of `UserCommand` subclasses, each containing
 - An command method (`execute()`)
 - State needed by the command



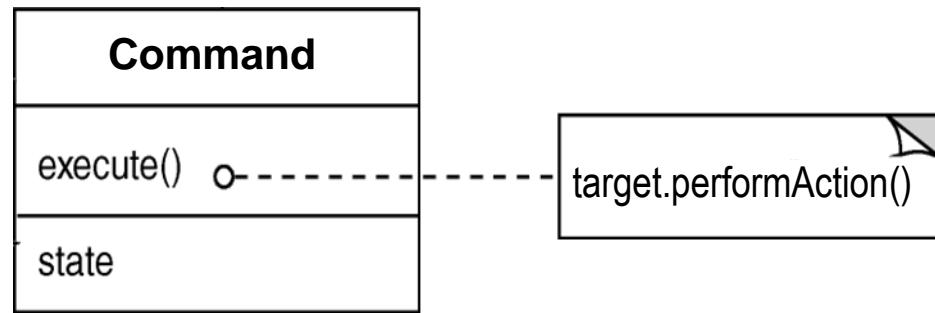
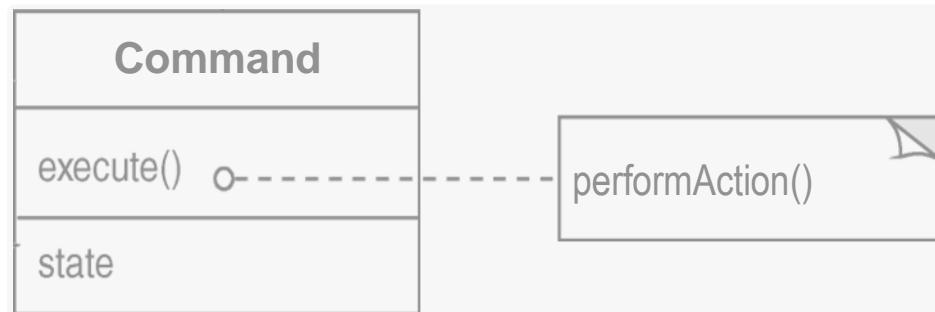
Solution: Encapsulate User Requests as Commands

- 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 performs a user-requested command on an expression tree when it's executed

Class methods

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

UserCommand Class Overview

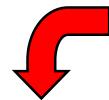
- Defines an abstract super class performs a user-requested command on an expression tree when it's executed

Class methods

`void execute()`

`void printValidCommands()`

These methods are
defined by subclasses



UserCommand Class Overview

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

Class methods

`void execute()`

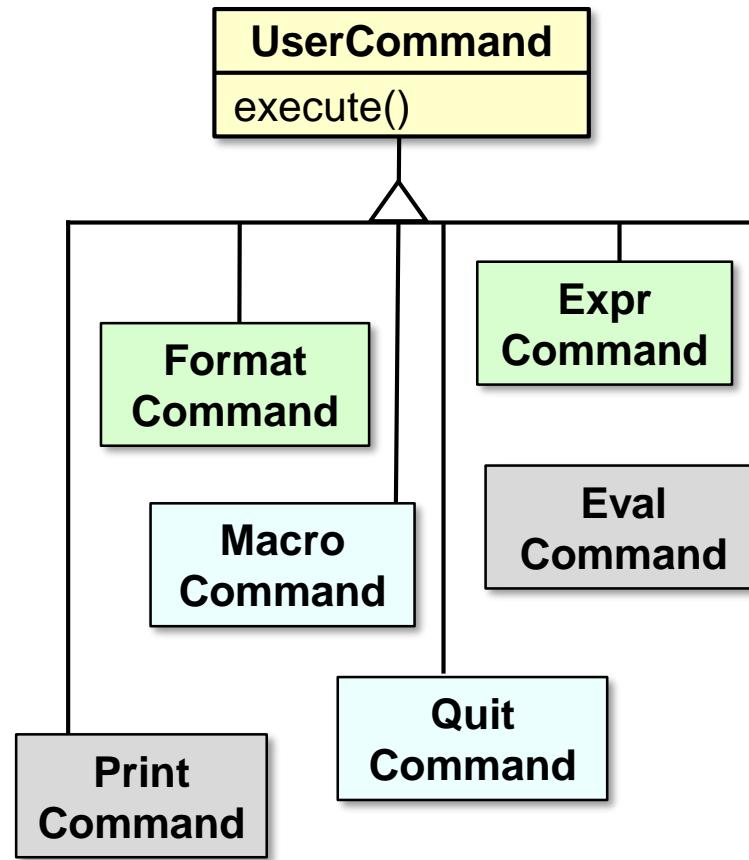
`void printValidCommands()`

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

Elements of the Command Pattern

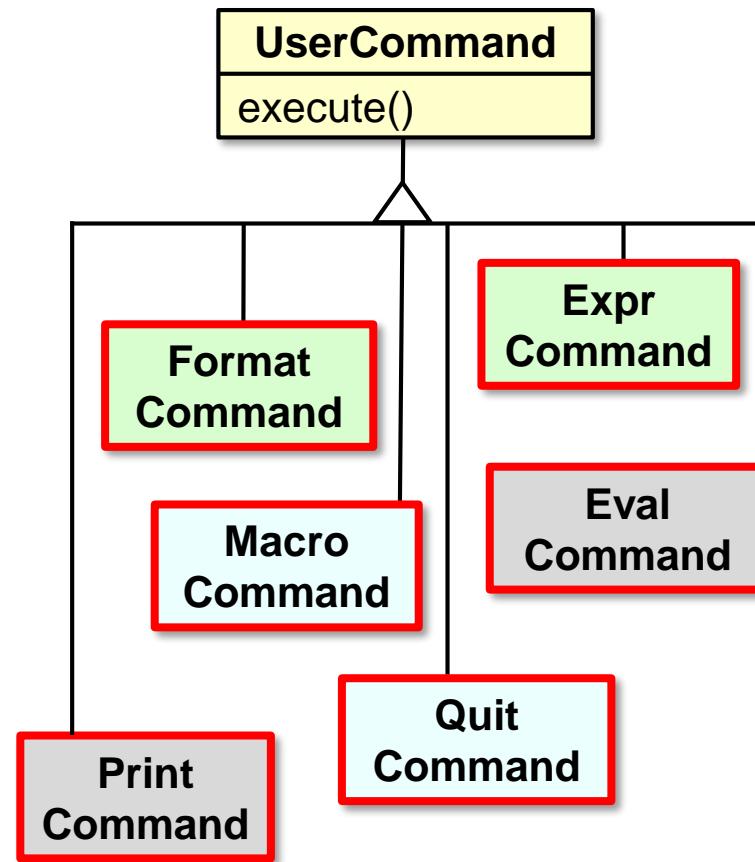
Intent

- Encapsulate the request for a service as an object



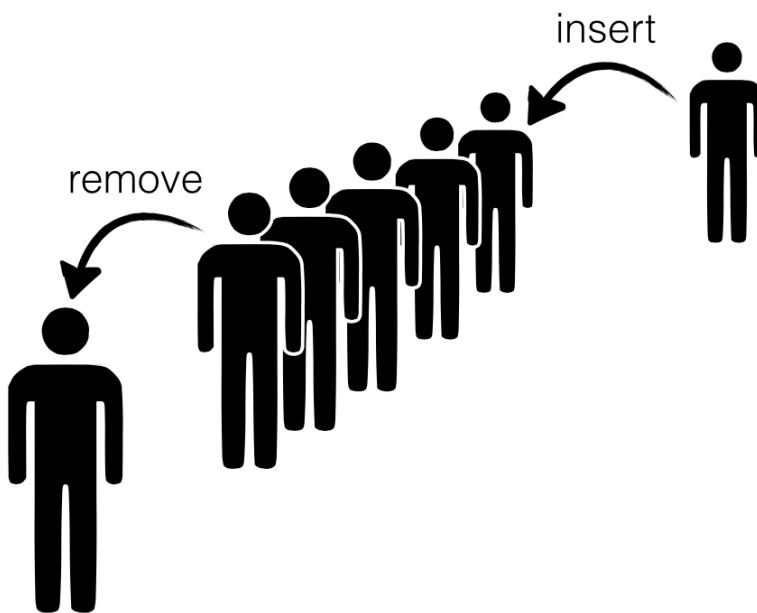
Applicability

- Want to parameterize objects with an action to perform



Applicability

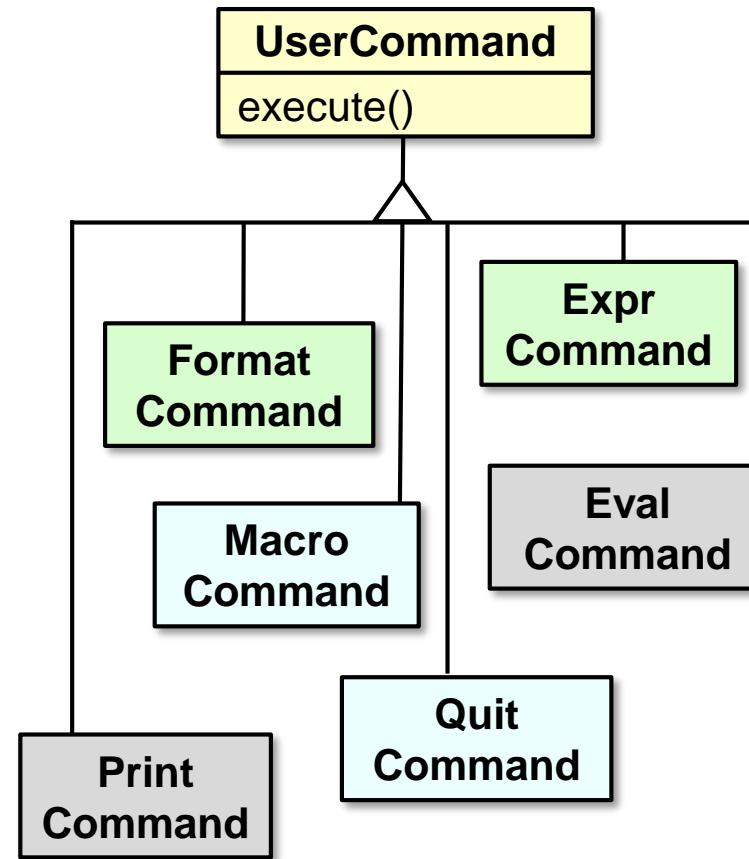
- Want to parameterize objects with an action to perform
- Want to specify, queue, & execute requests at different times



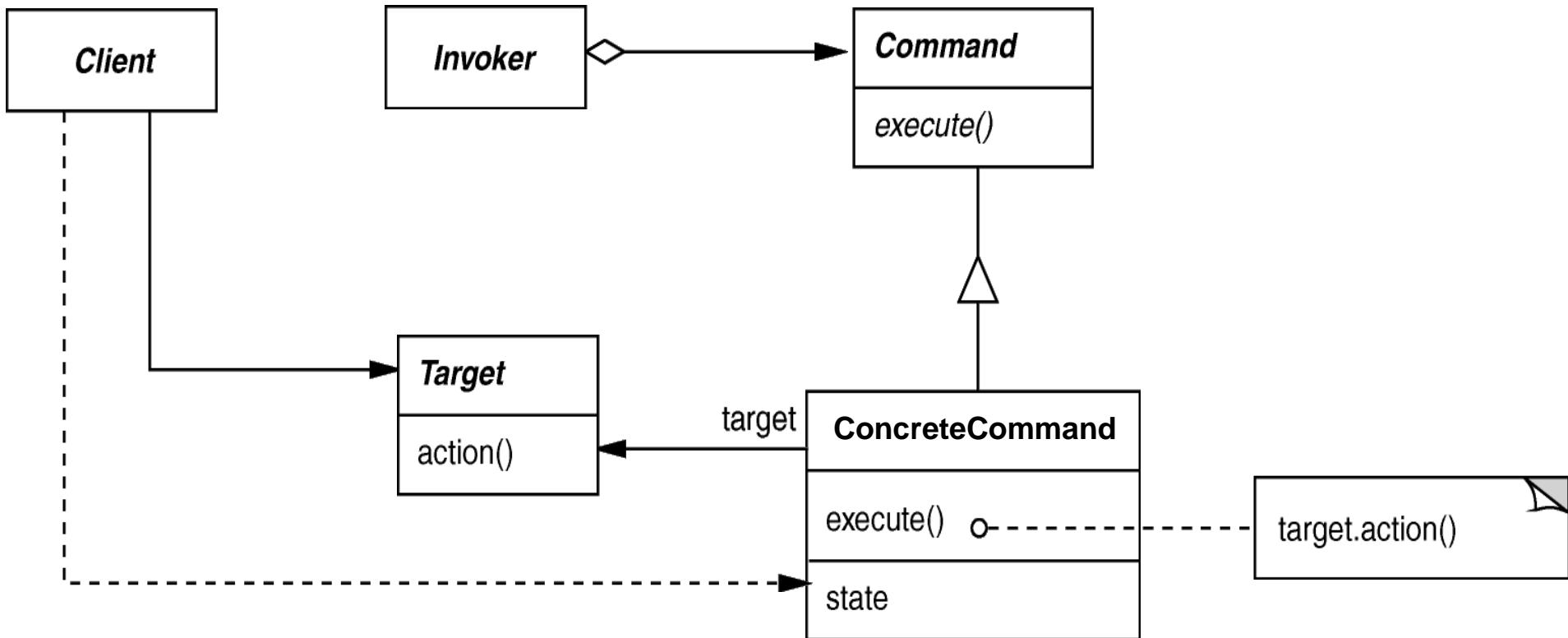
Applicability

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

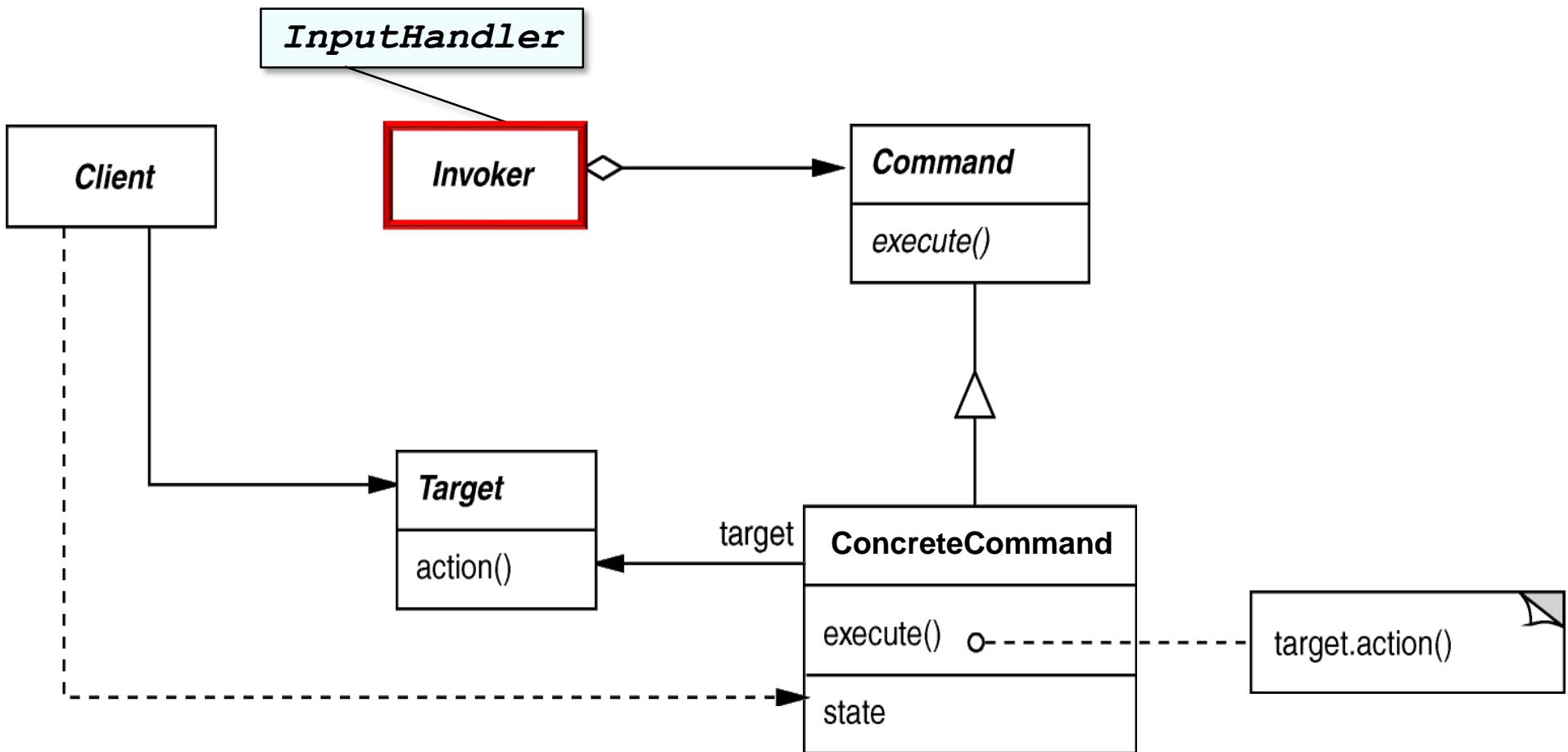
I NEED A
MULLIGAN!



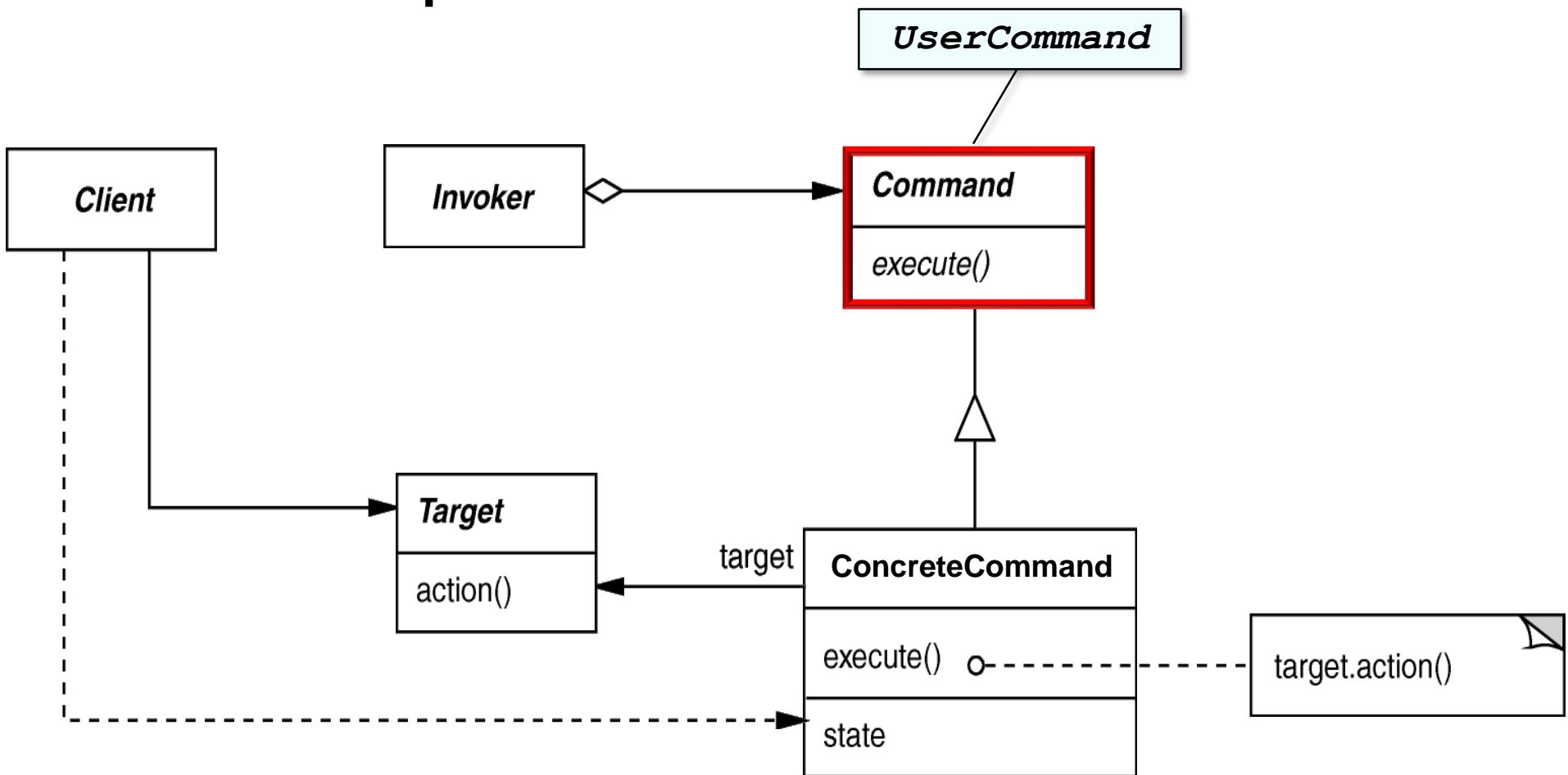
Structure & Participants



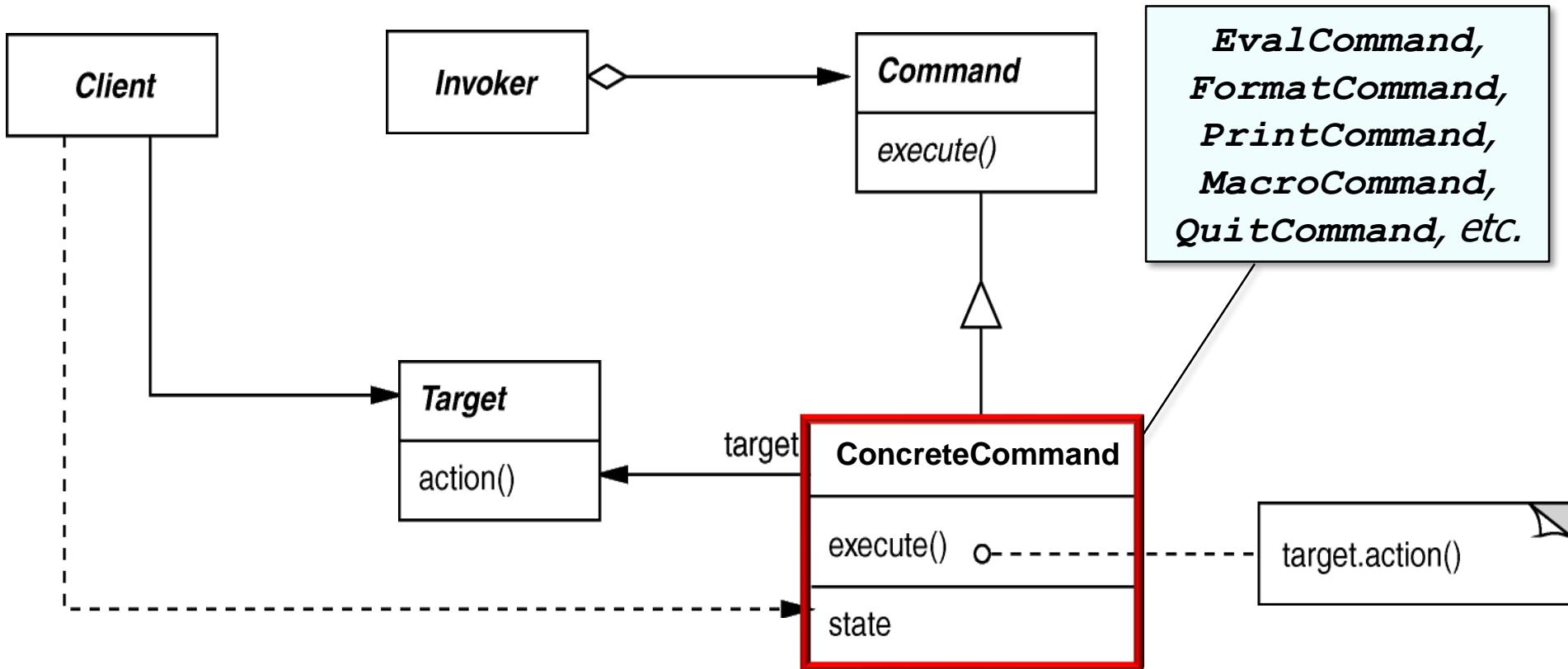
Structure & Participants



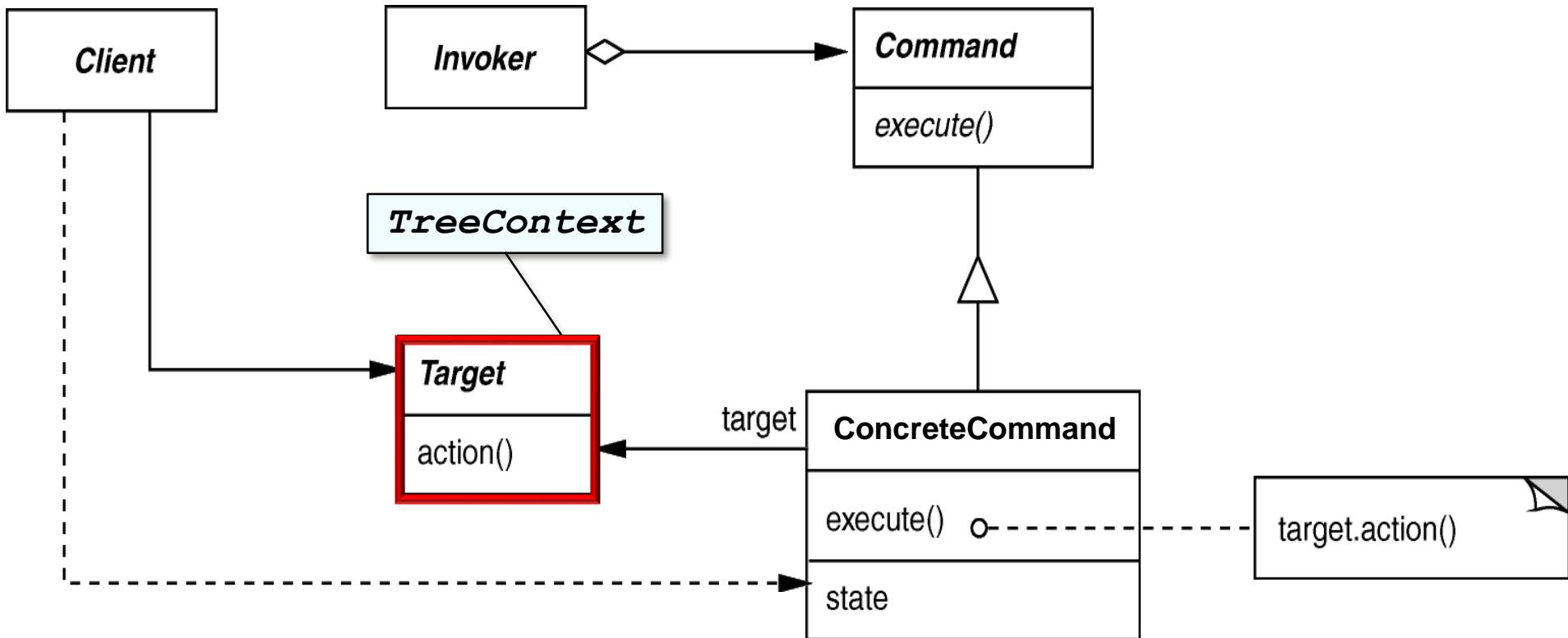
Structure & Participants



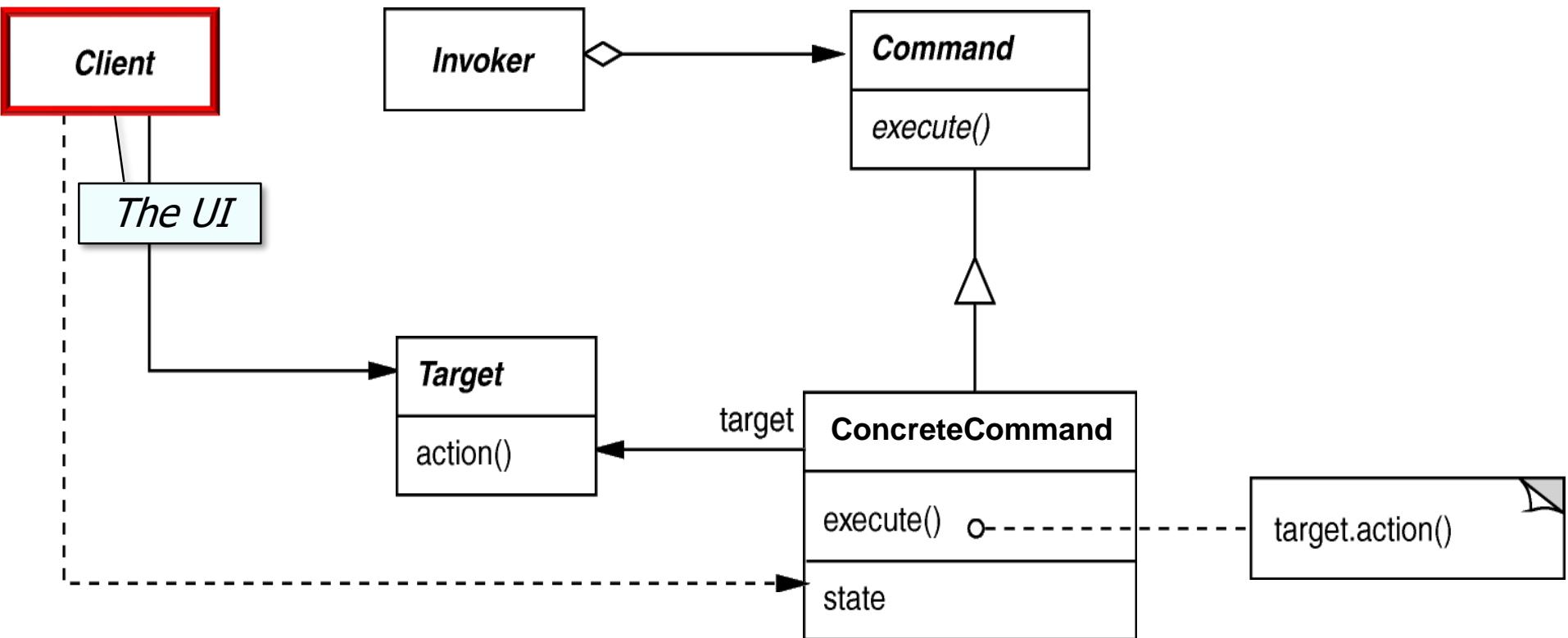
Structure & Participants



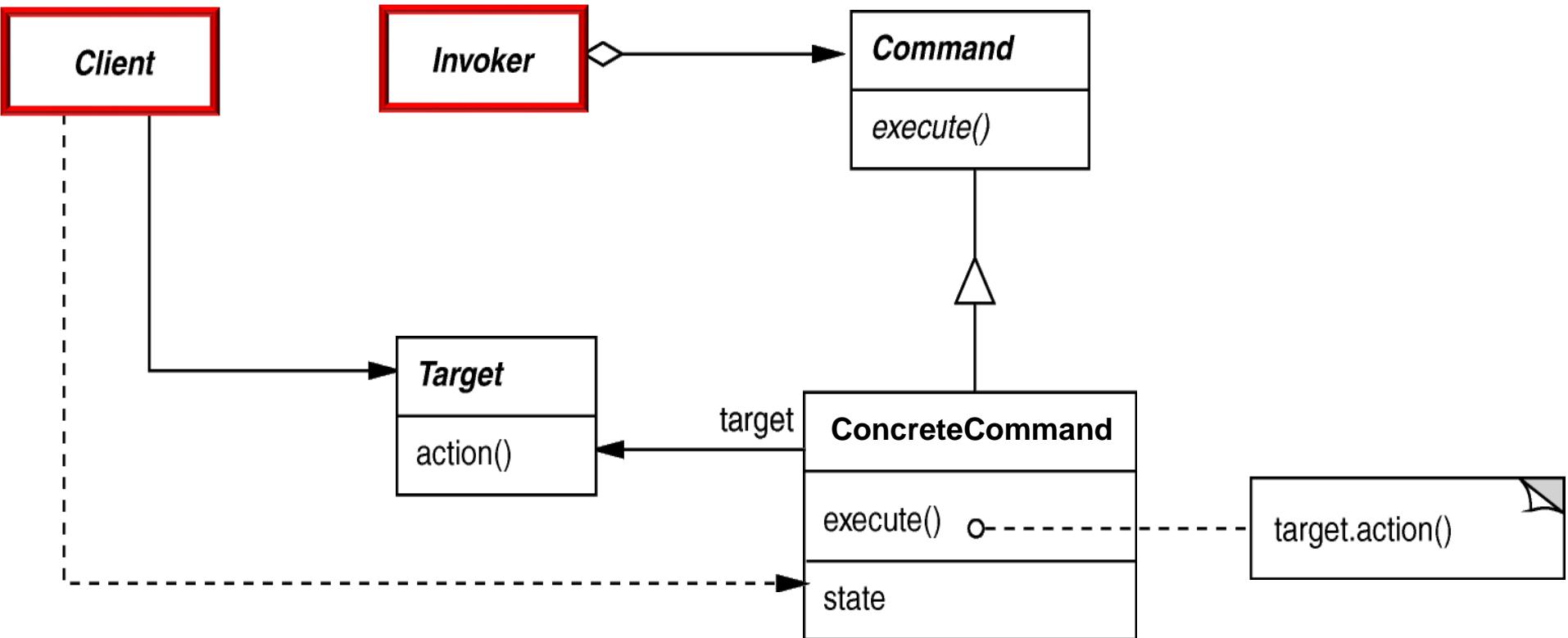
Structure & Participants



Structure & Participants



Structure & Participants



The **Client** & **Invoker** objects may be the same or different

Command example in Java

- Encapsulate execution of a command object that sets the desired expression
 - e.g., "1+2*3"

```
public class ExprCommand  
    extends UserCommand {  
  
    private String mExpr;  
  
    ExprCommand(TreeContext context,  
                String newexpr) {  
        super(context);  
        mExpr = newexpr;  
    }  
  
    public void execute() {  
        mTreeContext.makeTree(mExpr);  
    }  
}
```

See [ExpressionTree/CommandLine/src/expressiontree/commands](#)

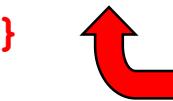
Command example in Java

- Encapsulate execution of a command object that sets the desired expression
 - e.g., "1+2*3"

```
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.makeTree(mExpr);  
    }  
}
```

Command example in Java

- Encapsulate execution of a command object that sets the desired expression
 - e.g., "1+2*3"

```
public class ExprCommand  
    extends UserCommand {  
  
    private String mExpr;  
  
    ExprCommand(TreeContext context,  
                String newexpr) {  
        super(context);  
        mExpr = newexpr;  
    }  
  
    public void execute() {  
        mTreeContext.makeTree(mExpr);  
    }  
  
      
    Provide appropriate TreeContext  
& requested expression
```

Command example in Java

- Encapsulate execution of a command object that sets the desired expression
 - e.g., "1+2*3"

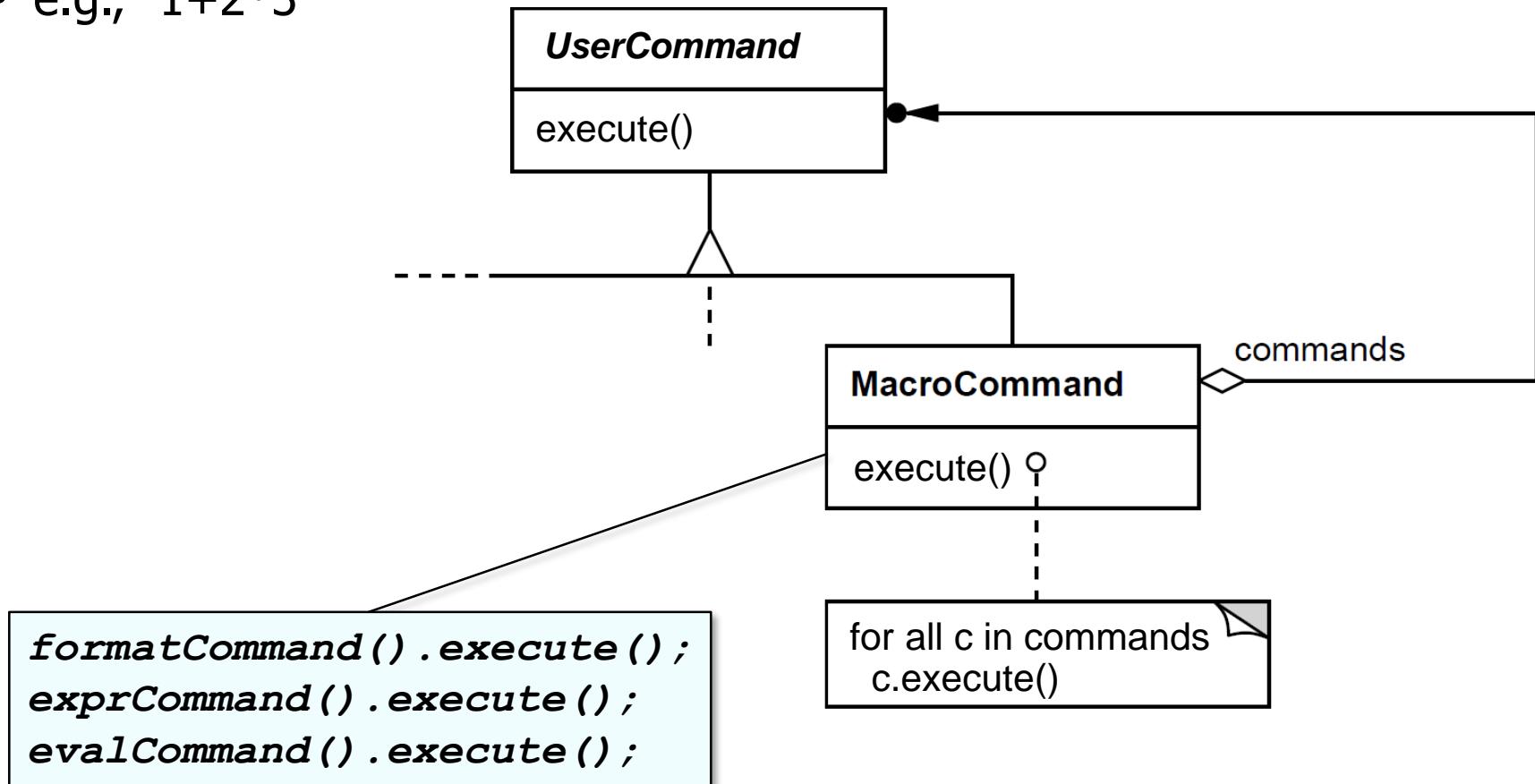
```
public class ExprCommand  
    extends UserCommand {  
  
    private String mExpr;  
  
    ExprCommand(TreeContext context,  
                String newexpr) {  
        super(context);  
        mExpr = newexpr;  
    }  
  
    public void execute() {  
        mTreeContext.makeTree(mExpr);  
    }  
}
```



Create desired expression tree

Command example in Java

- Encapsulate execution of a sequence of commands as an object, which is used to implement “succinct mode”
 - e.g., "1+2*3"



See [ExpressionTree/CommandLine/src/expressiontree/commands](#)

Command example in Java

- Encapsulate execution of a sequence of commands as an object, which is used to implement “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 execution of a sequence of commands as an object, which is used to implement “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

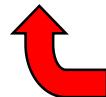
- Encapsulate execution of a sequence of commands as an object, which is used to implement “succinct mode”

```
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 execution of a sequence of commands as an object, which is used to implement “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();  
    }  
    ...
```



Executes a sequence of commands to implement “succinct mode”

Command example in Java

- Encapsulate execution of a sequence of commands as an object, which is used to implement “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 “succinct mode”

Consequences

+ Abstracts executor of a service

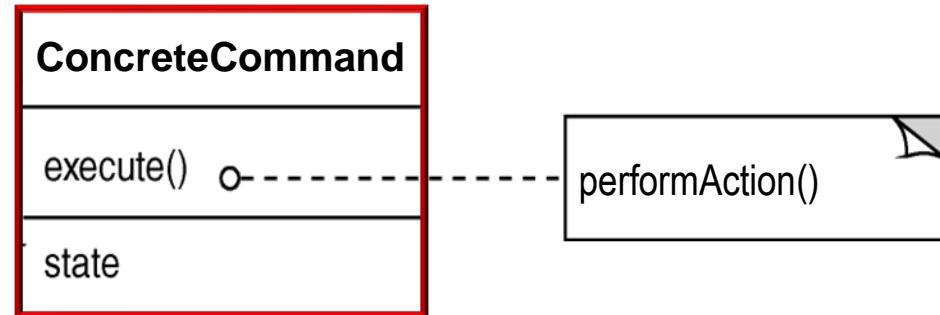
- Makes programs more modular & flexible



Consequences

+ Abstracts executor of a service

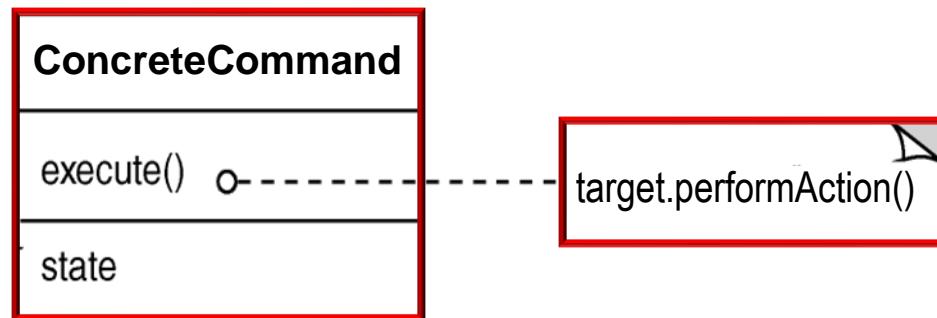
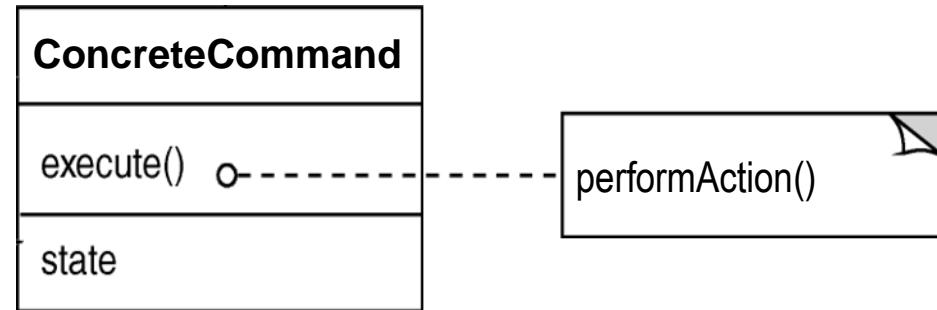
- Makes programs more modular & flexible, e.g.
 - Can bundle state & behavior into an object



Consequences

+ Abstracts executor of a service

- Makes programs more modular & flexible, e.g.
 - Can bundle state & behavior into an object
 - Can forward behavior to other objects

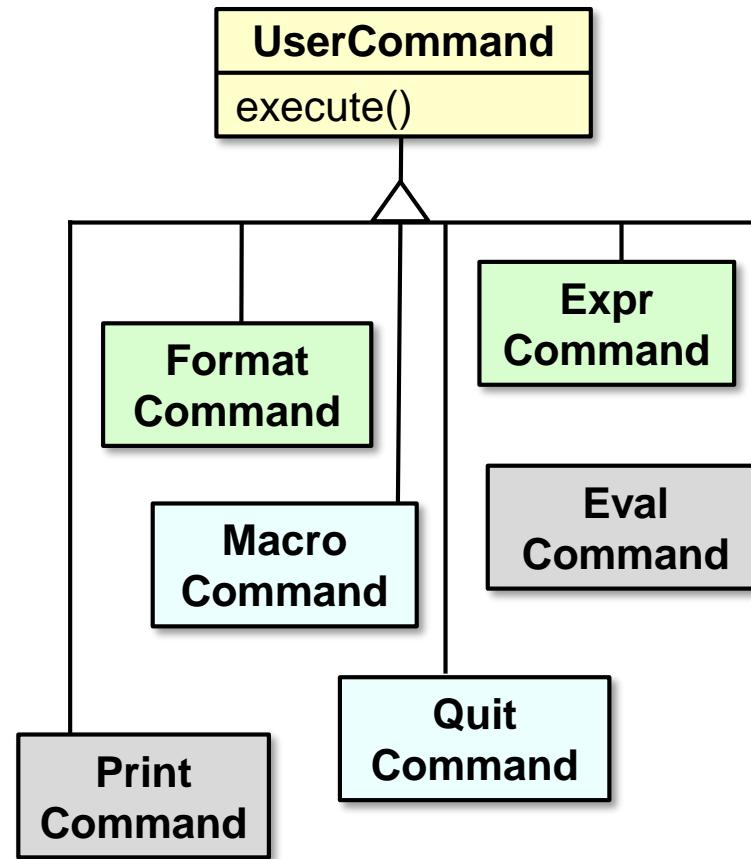


See upcoming lesson on the *State* pattern for an example of forwarding

Consequences

+ Abstracts executor of a service

- Makes programs more modular & flexible, e.g.
 - Can bundle state & behavior into an object
 - Can forward behavior to other objects
 - Can extend behavior via subclassing



Consequences

+ Abstracts executor of a service

- Makes programs more modular & flexible, e.g.
 - Can bundle state & 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 executor of a service

- Makes programs more modular & flexible, e.g.
 - Can bundle state & 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);
```



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

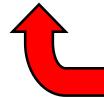
```
executeCommand(command);
```

Consequences

+ Abstracts executor of a service

- Makes programs more modular & flexible, e.g.
 - Can bundle state & 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);
```

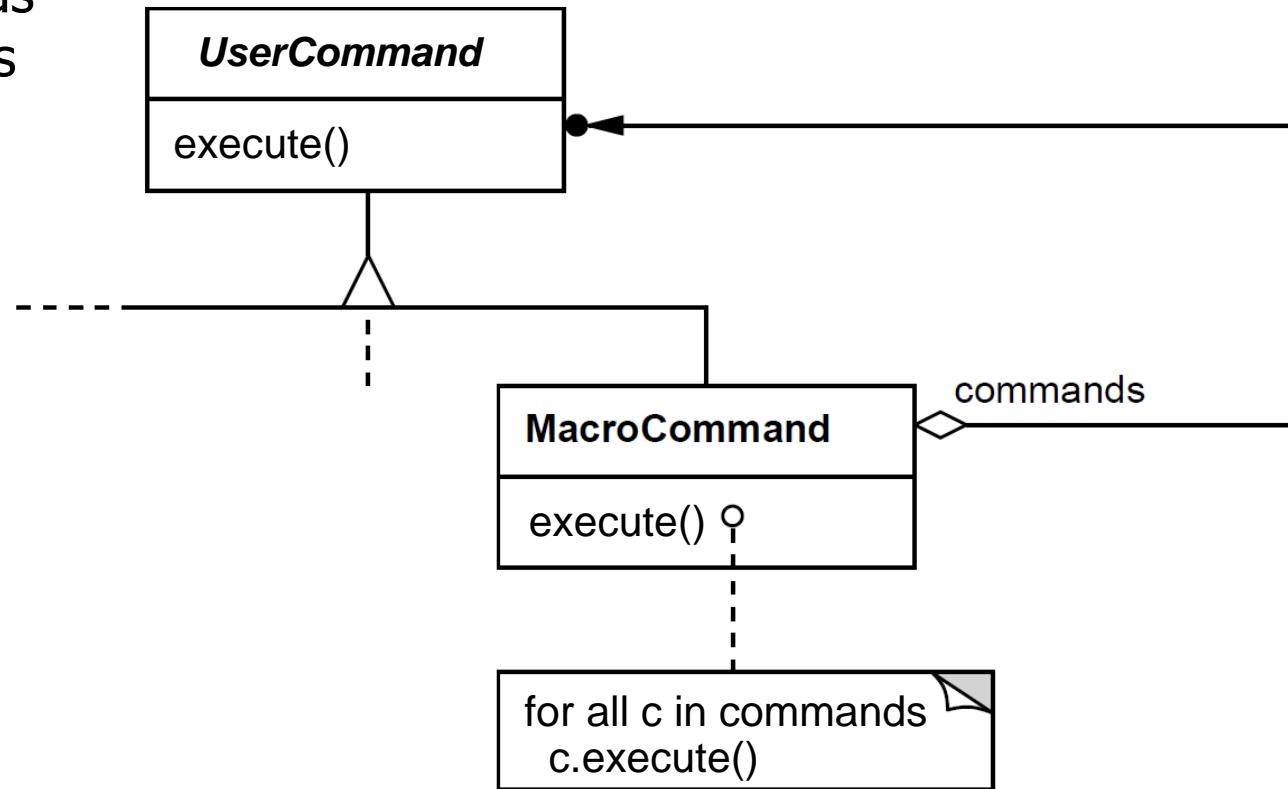


Call a hook method & pass a command to execute

Consequences

+ Abstracts executor of a service

+ Composition yields macro-commands



Consequences

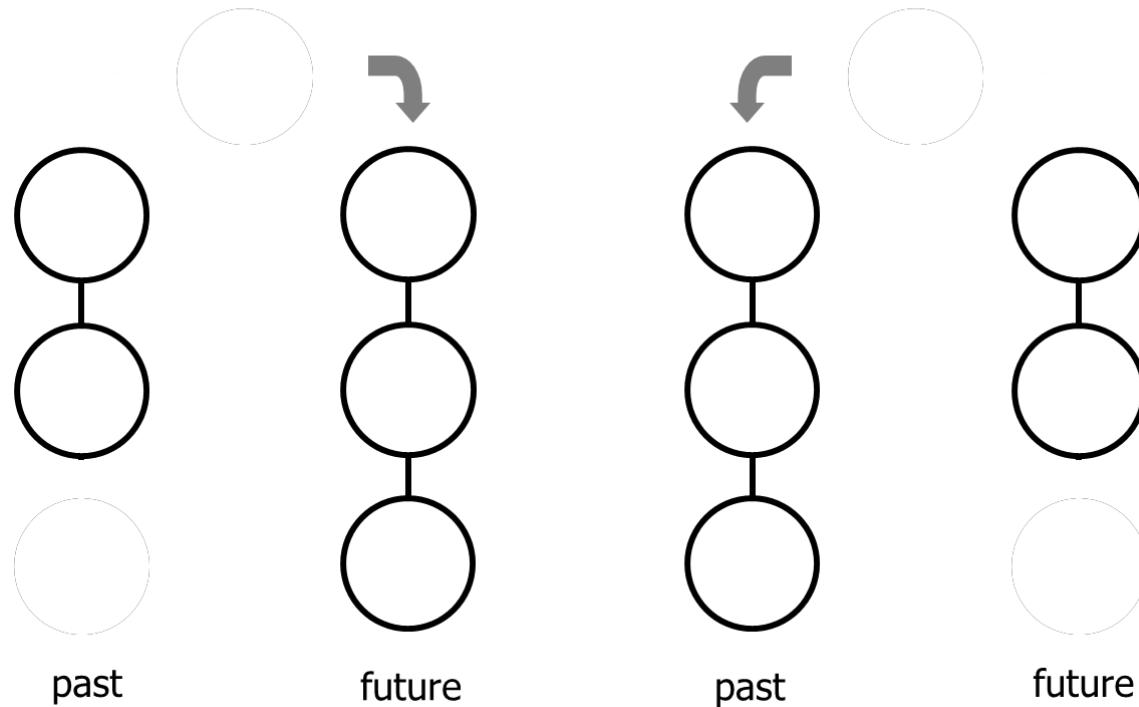
- + Abstracts executor of a service
- + Composition yields macro-commands
- + Supports arbitrary-level undo-redo

Undo:

`unexecute()`

Redo:

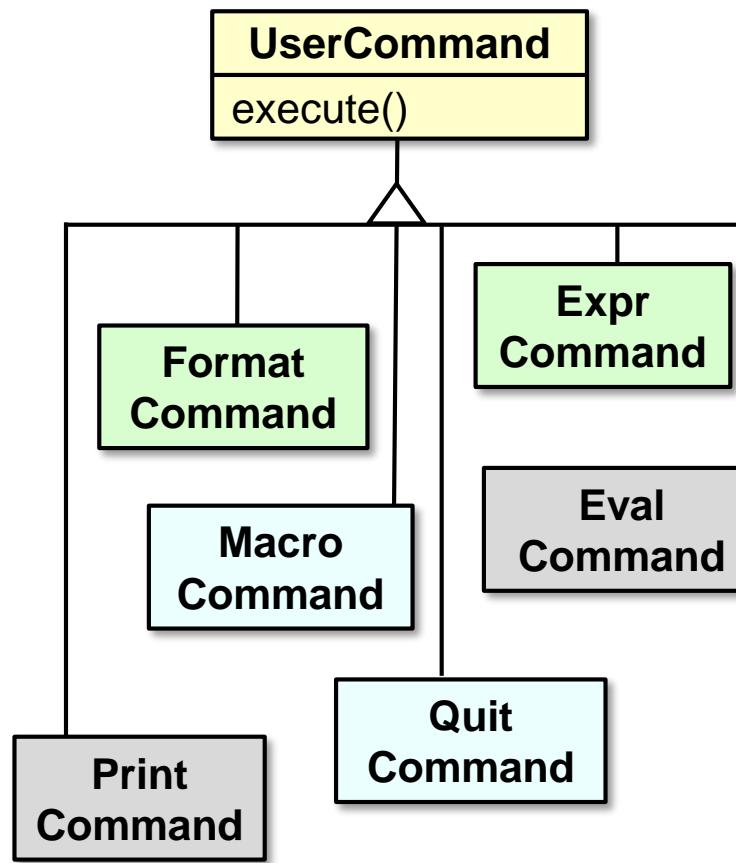
`execute()`



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

Consequences

- Might result in lots of trivial command subclasses

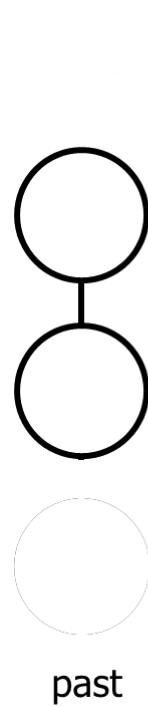


Consequences

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

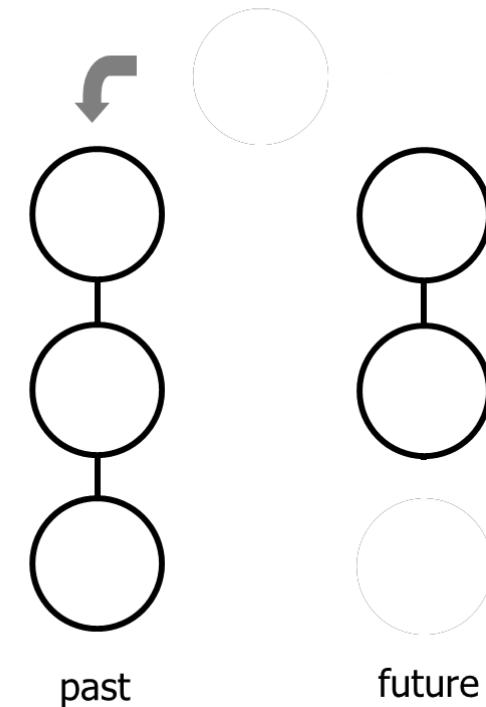
Undo:

`unexecute()`



Redo:

`execute()`

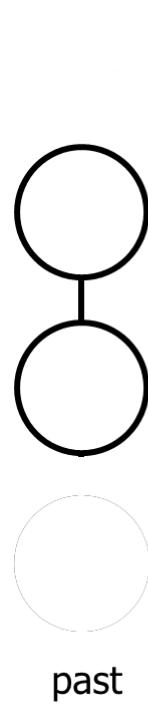


Implementation

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

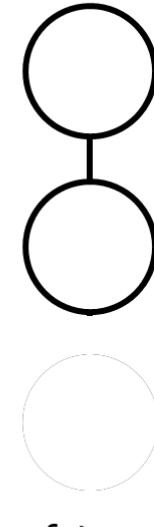
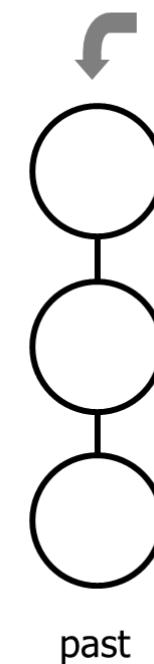
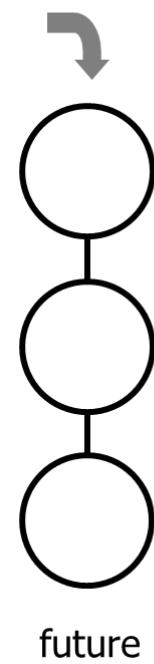
Undo:

`unexecute()`



Redo:

`execute()`



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:

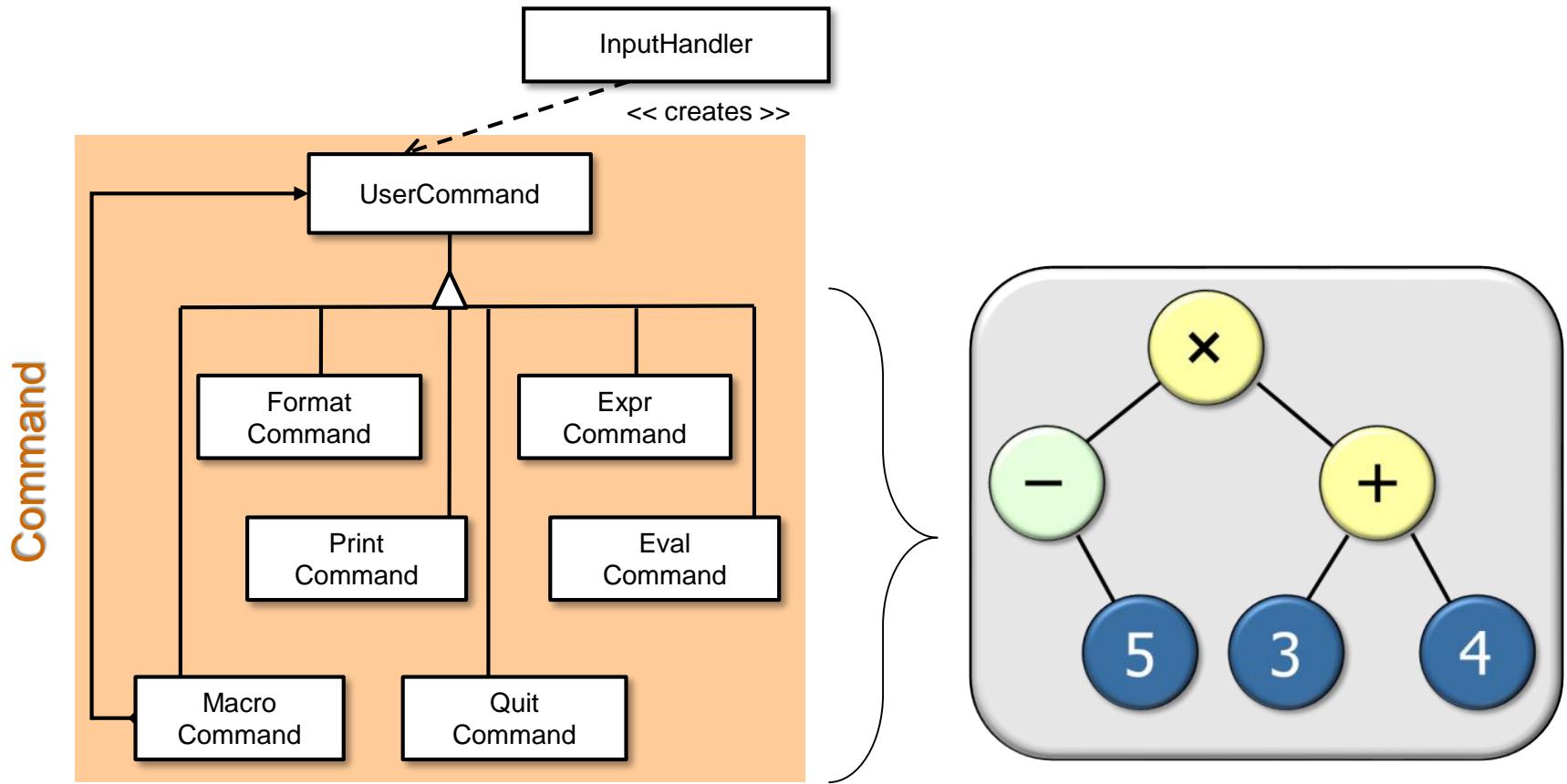
[AsyncBoxView.ChildState](#), [FutureTask](#),
[RenderableImageProducer](#), [SwingWorker](#), [Thread](#), [TimerTask](#)

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.

Summary of the Command Pattern

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



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

**End of the
Command Pattern**

The Factory Method Pattern

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



Professor of Computer Science

Institute for Software
Integrated Systems

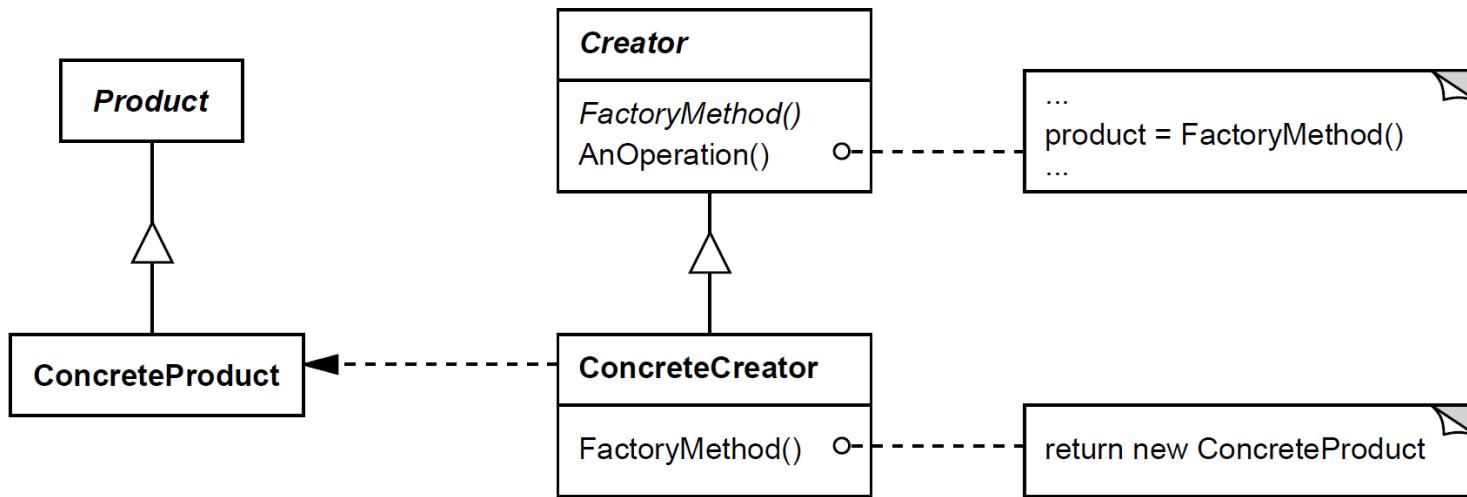
Vanderbilt University
Nashville, Tennessee, USA



Learning Objectives

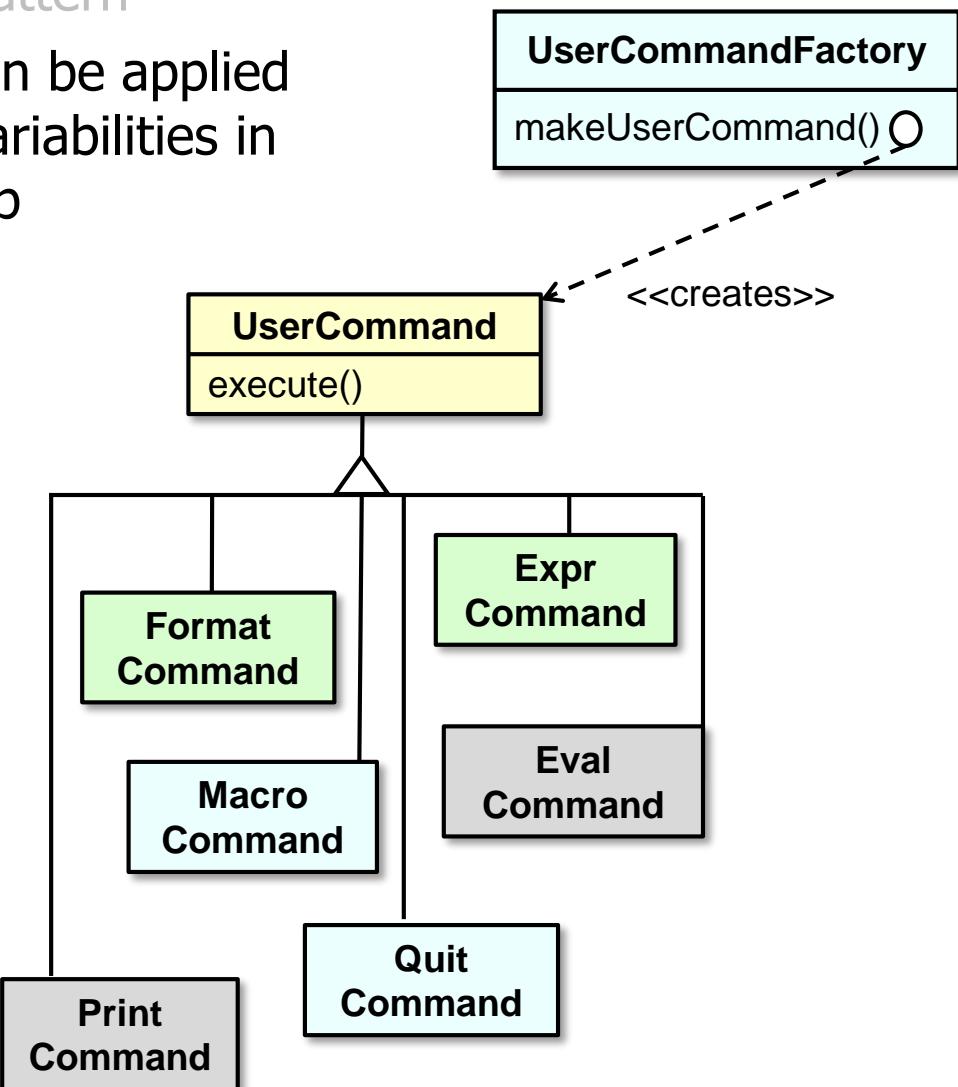
Learning Objectives

- Understand the *Factory Method* pattern



Learning Objectives

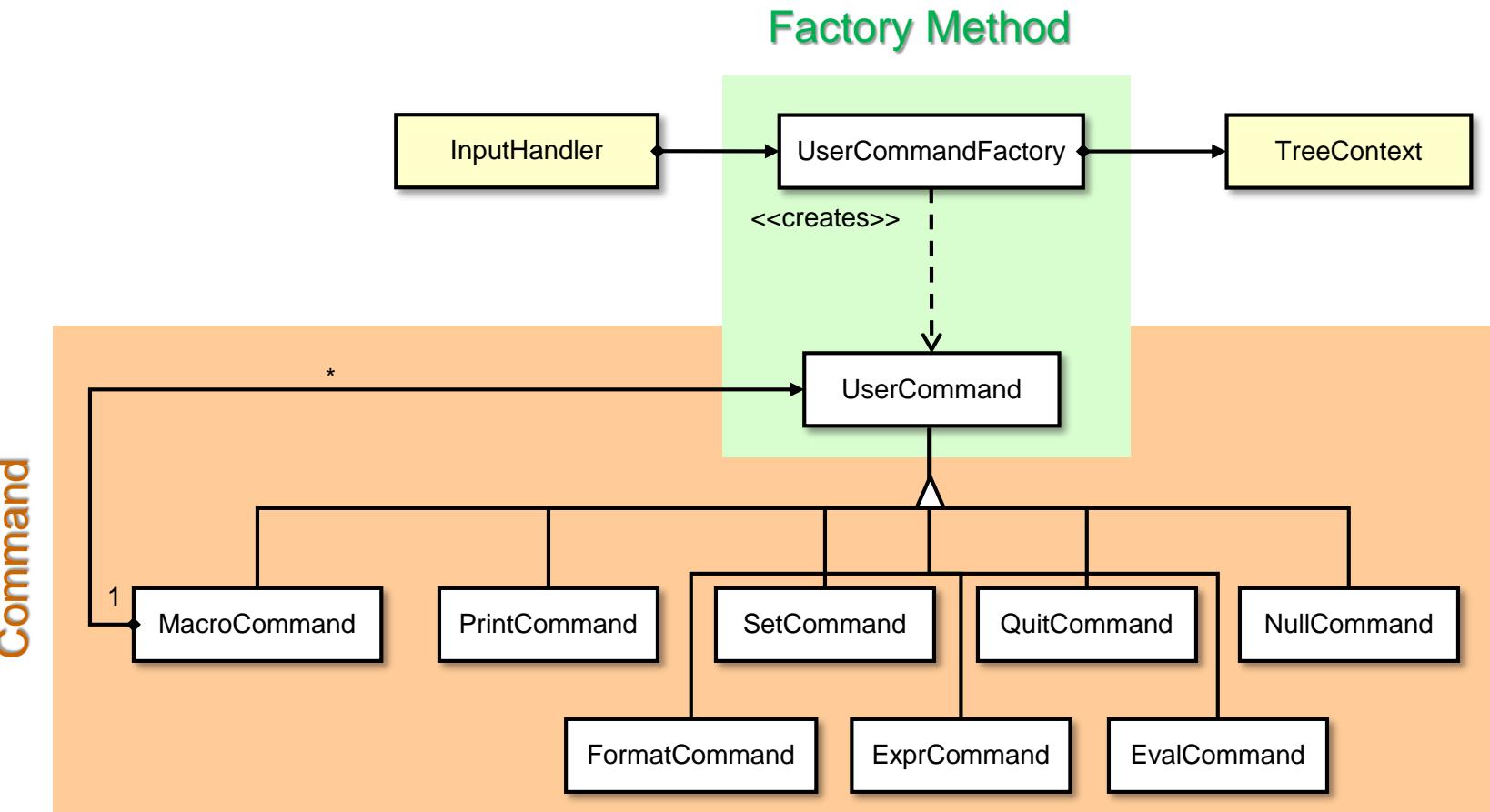
- Understand the *Factory Method* pattern
- Recognize how *Factory Method* can be applied to enable extensible creation of variabilities in the expression tree processing app



Motivating the Need for the Factory Method Pattern in the Expression Tree App

A Pattern for Abstracting Object Creation

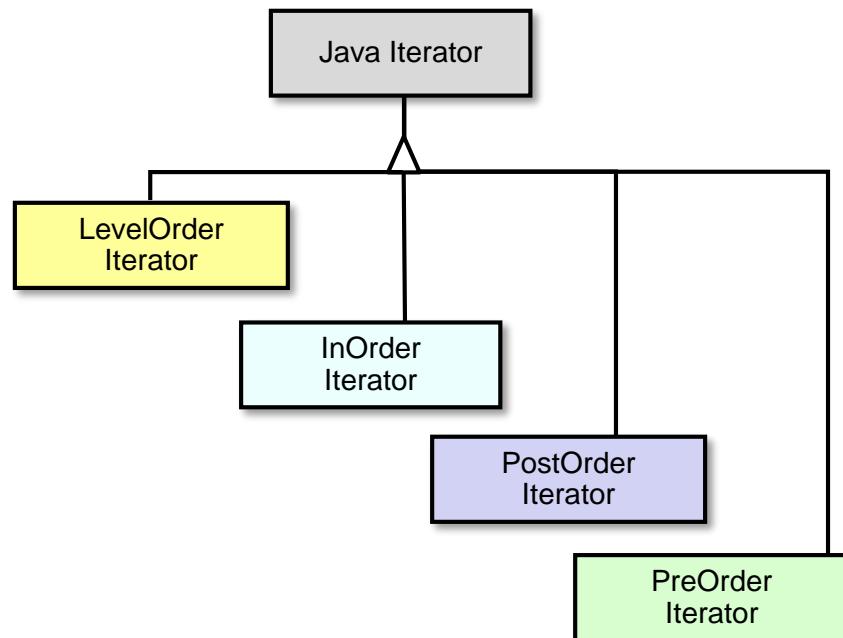
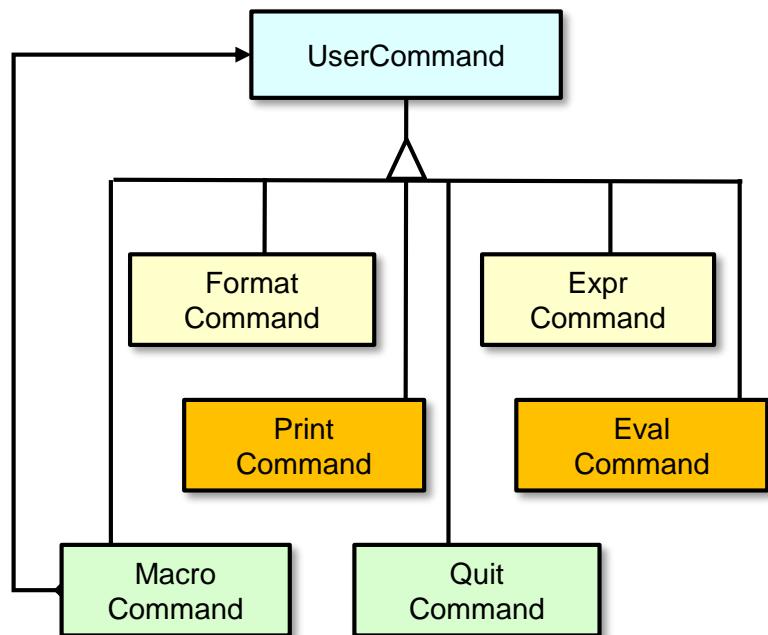
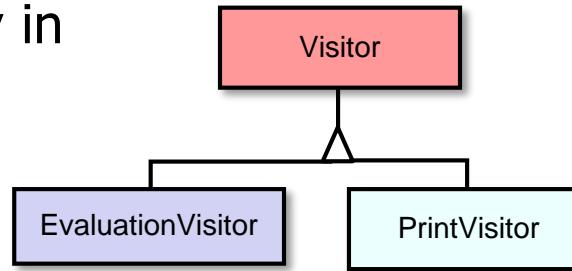
Purpose: Enable the extensible creation of variabilities, such as commands, iterators, & visitors



Factory Method decouples the creation of objects from their subsequent use

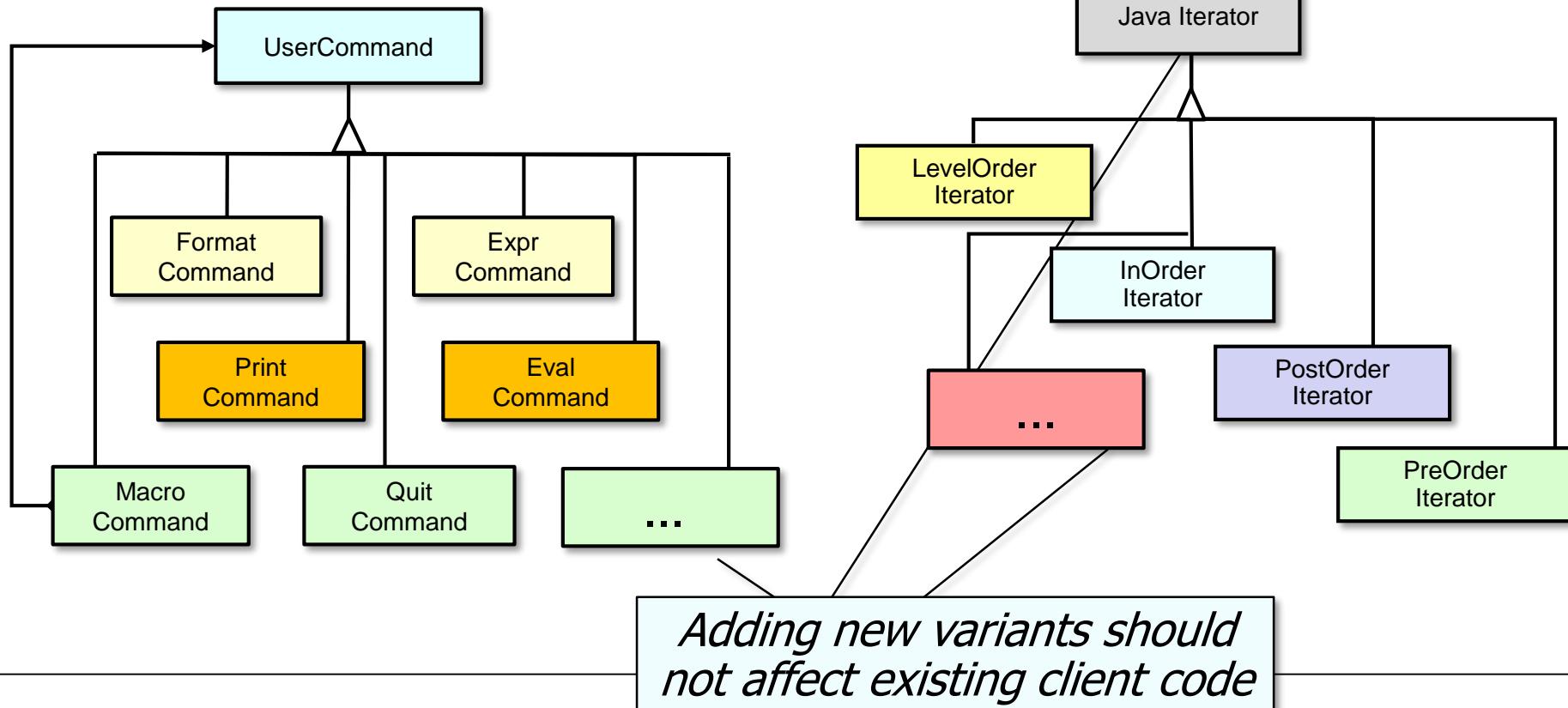
Context: OO Expression Tree Processing App

- There are many points of variability in the expression tree processing app
 - e.g., user commands, traversal strategies, & visitor operations applied on an expression tree



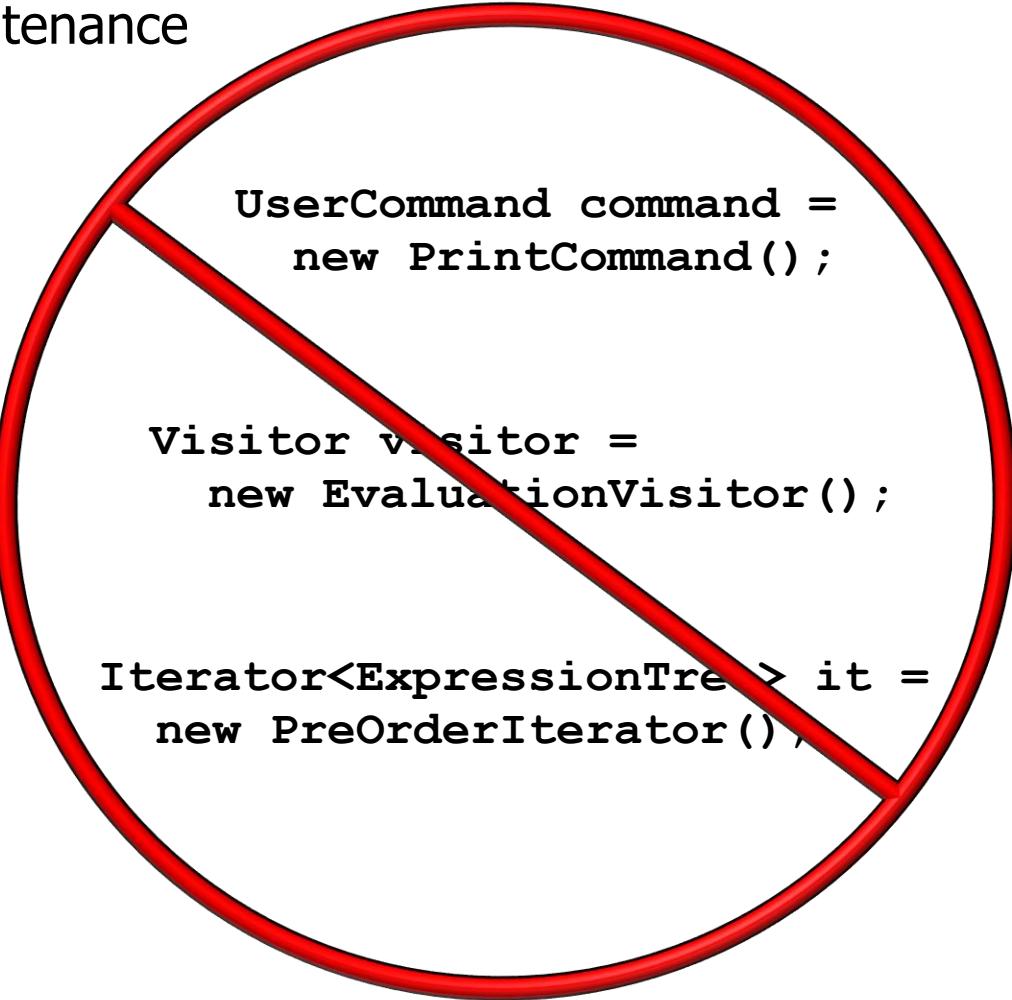
Context: OO Expression Tree Processing App

- There are many points of variability in the expression tree processing app
 - e.g., user commands, traversal strategies, & visitor operations applied on an expression tree



Problem: Inflexible Creation of Variabilities

- Tightly coupling the creation of variabilities with client code is problematic
 - e.g., hard-coding dependencies on specific subclasses can complicate maintenance & impede extensibility



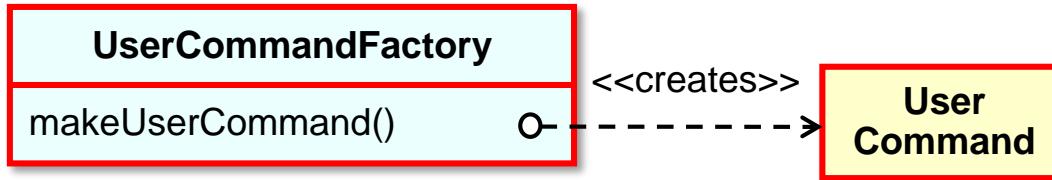
```
UserCommand command =  
    new PrintCommand();
```

```
Visitor visitor =  
    new EvaluationVisitor();
```

```
Iterator<ExpressionTree> it =  
    new PreOrderIterator();
```

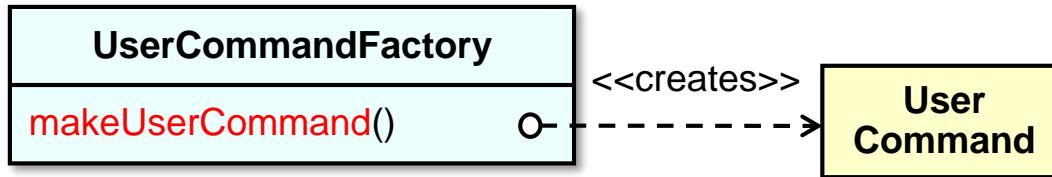
Solution: Abstract Creation of Objects

- Define a **UserCommandFactory** class whose **makeUserCommand()** factory method creates a **UserCommand** object



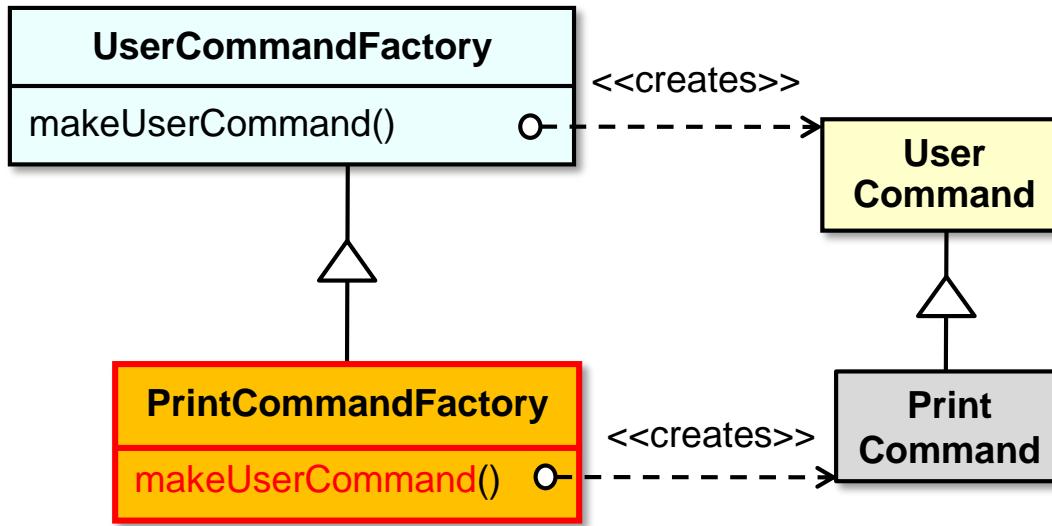
Solution: Abstract Creation of Objects

- Have the `makeUserCommand()` factory method implement the appropriate subclass of `UserCommand`



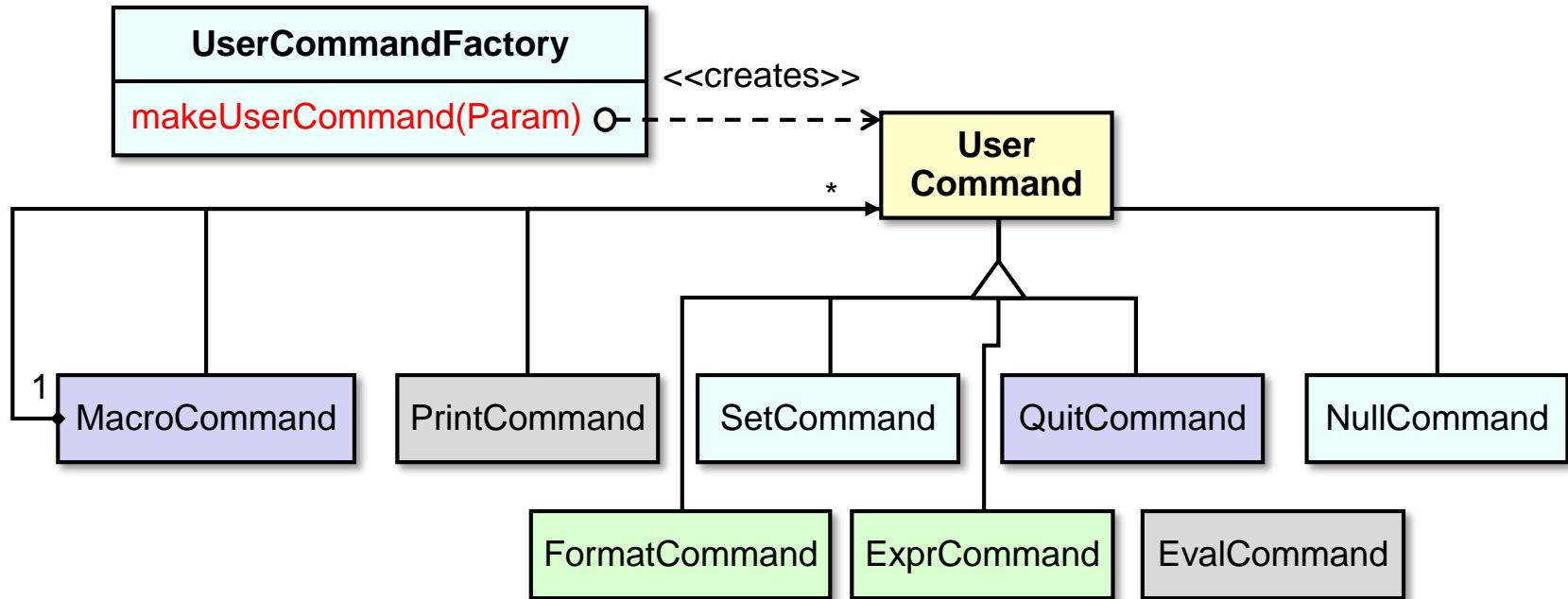
Solution: Abstract Creation of Objects

- Have the `makeUserCommand()` factory method implement the appropriate subclass of `UserCommand`, e.g.
 - Subclass `UserCommandFactory` & override the factory method `makeUserCommand()`



Solution: Abstract Creation of Objects

- Have the `makeUserCommand()` factory method implement the appropriate subclass of `UserCommand`, e.g.
 - Subclass `UserCommandFactory` & override the factory method `makeUserCommand()`



- Or pass a parameter to the `makeUserCommand()` factory method & use it to create the appropriate `UserCommand` subclass objects

UserCommandFactory Class Overview

- Create the command corresponding to user input

Class methods

UserCommand makeUserCommand(String inputString)

...

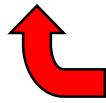
UserCommandFactory Class Overview

- Create the command corresponding to user input

Class methods

UserCommand **makeUserCommand**(String inputString)

...



This is a factory method

UserCommandFactory Class Overview

- Create the command corresponding to user input

Class methods

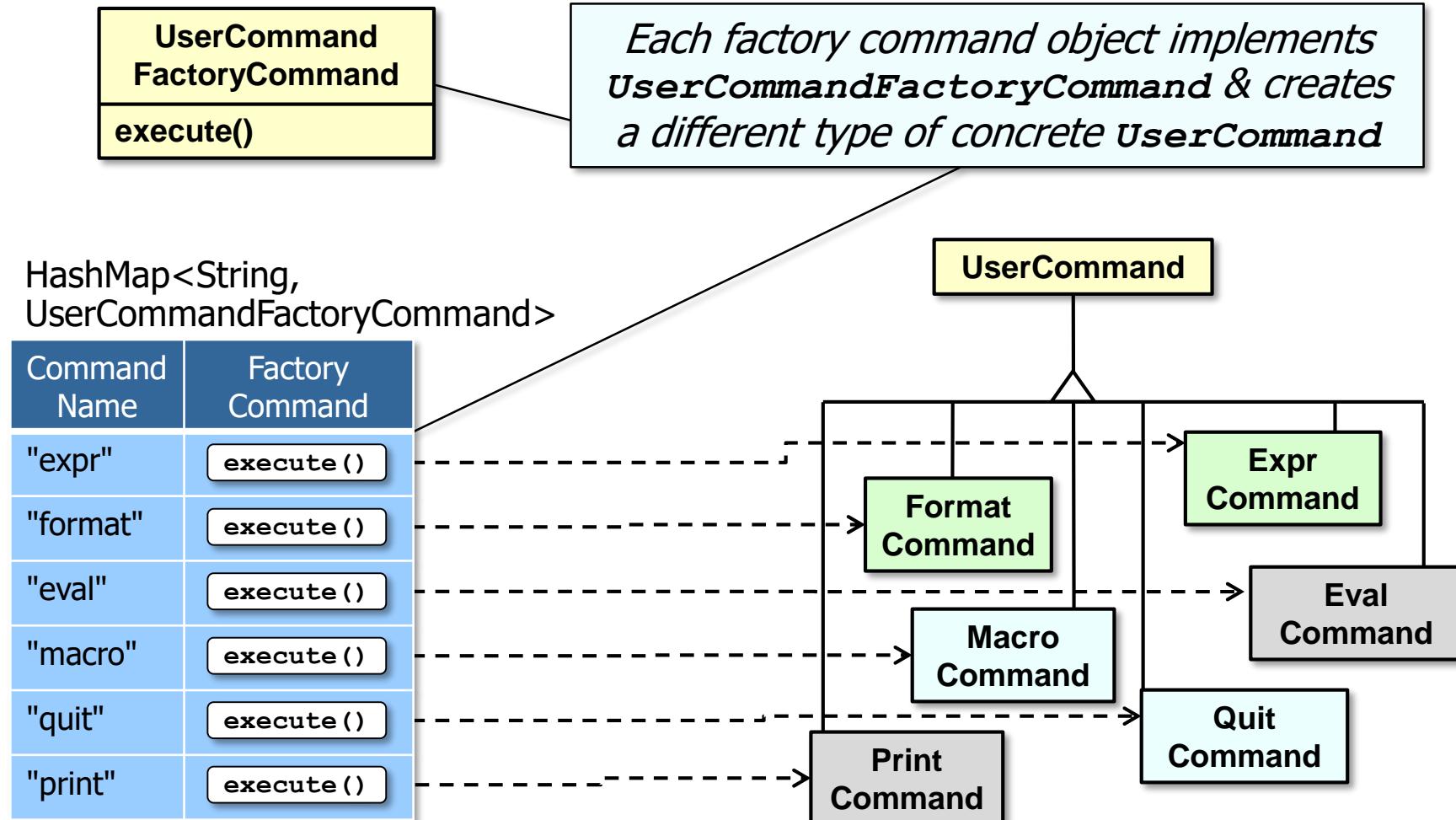
UserCommand makeUserCommand(String inputString)

...

- **Commonality:** Provides a common interface to create commands
- **Variability:** Implementations of expression tree command factory methods can vary depending on the requested commands

UserCommandFactory Class Overview

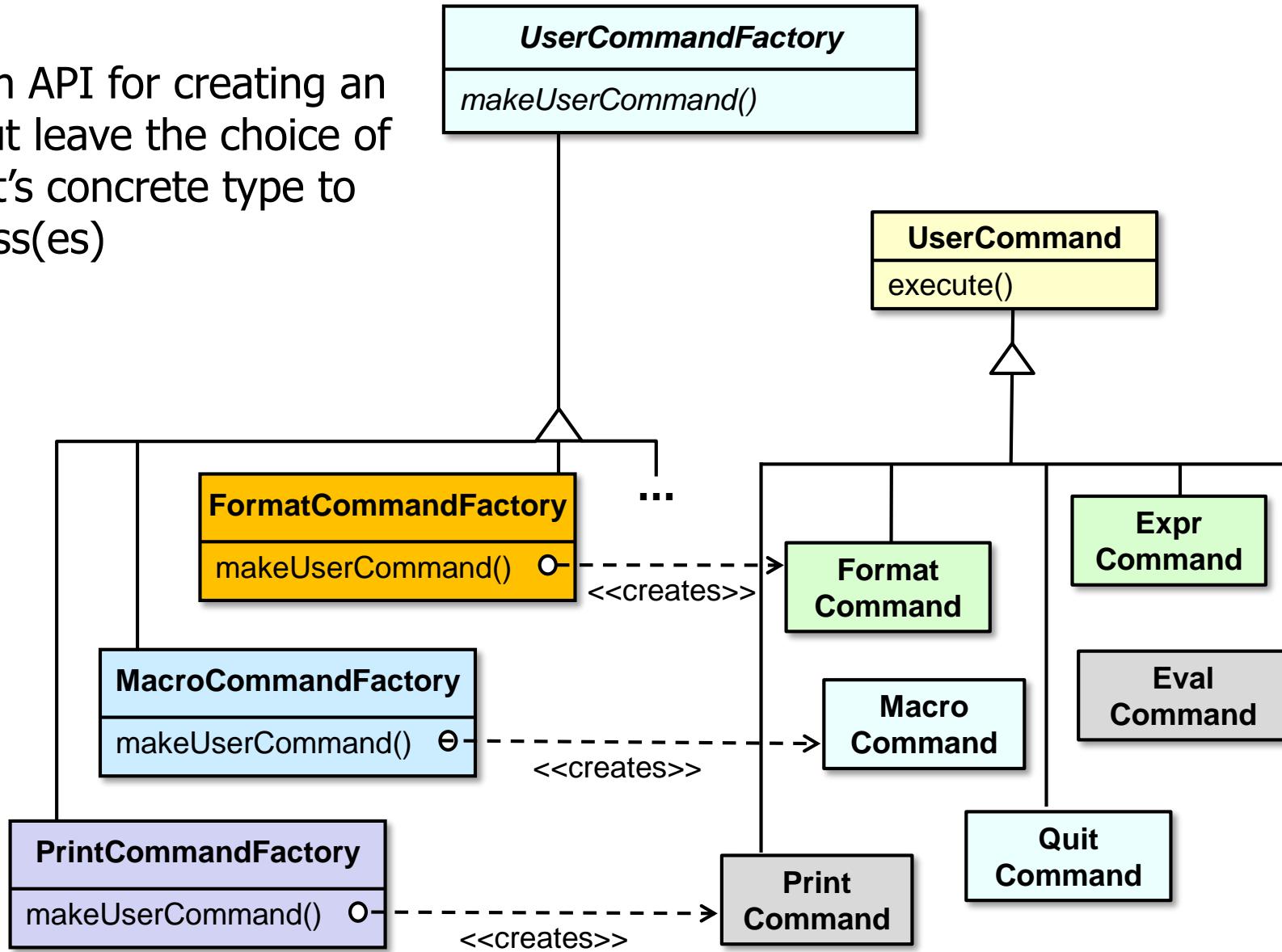
- Create the command corresponding to user input



Elements of the Factory Method Pattern

Intent

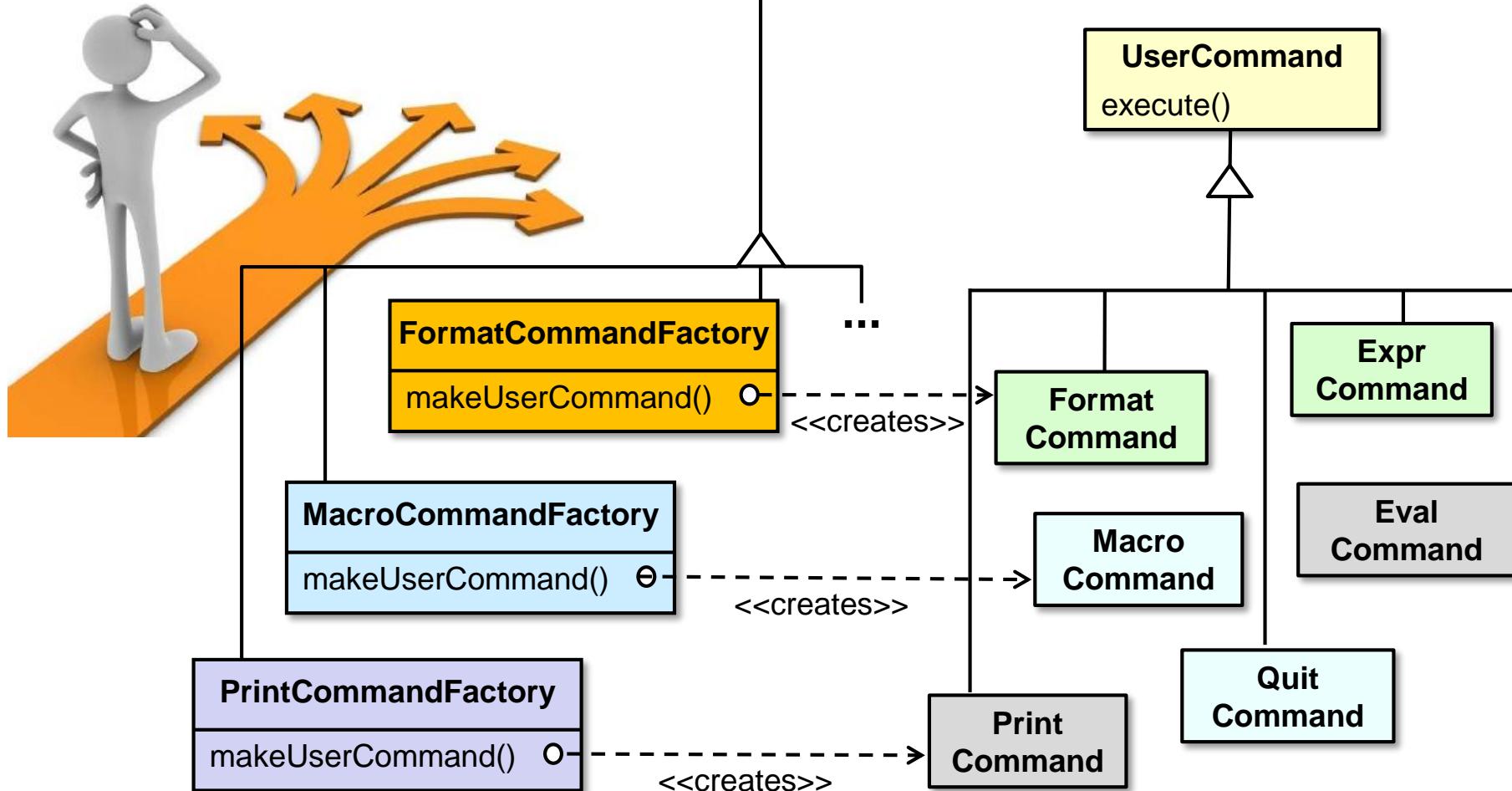
- Provide an API for creating an object, but leave the choice of the object's concrete type to its subclass(es)



See en.wikipedia.org/wiki/Factory_method_pattern

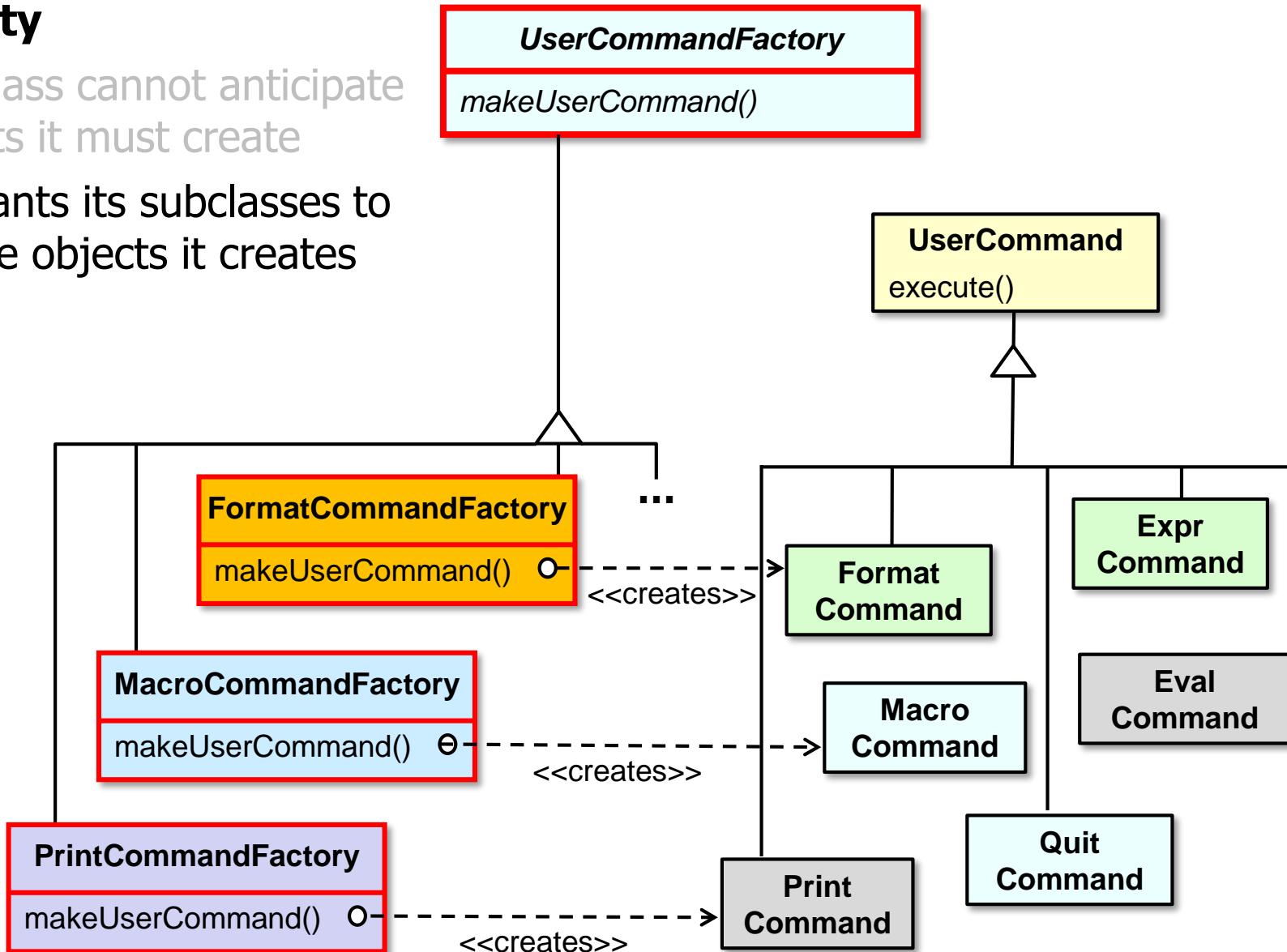
Applicability

- When a class cannot anticipate the objects it must create



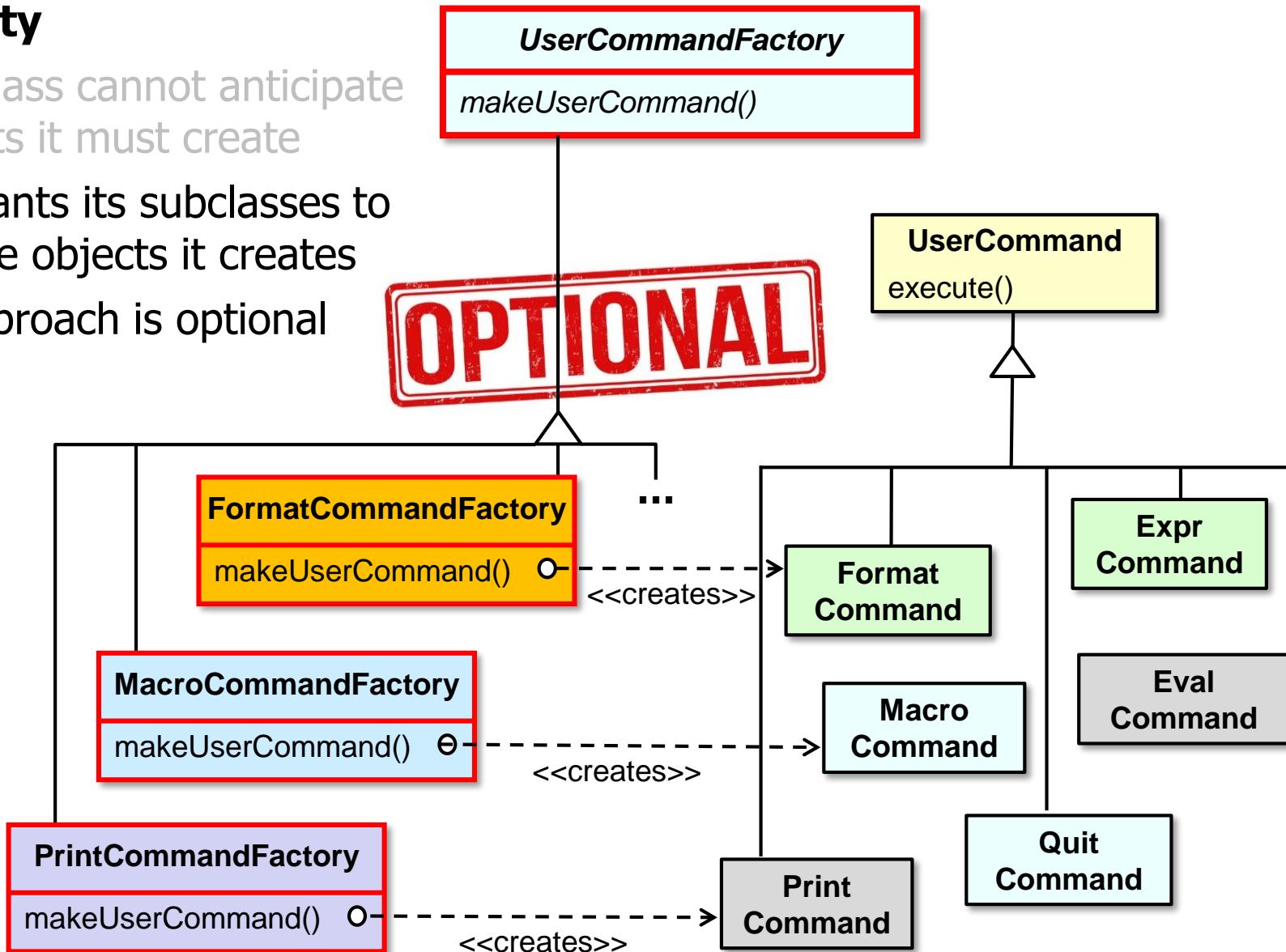
Applicability

- When a class cannot anticipate the objects it must create
- A class wants its subclasses to specify the objects it creates



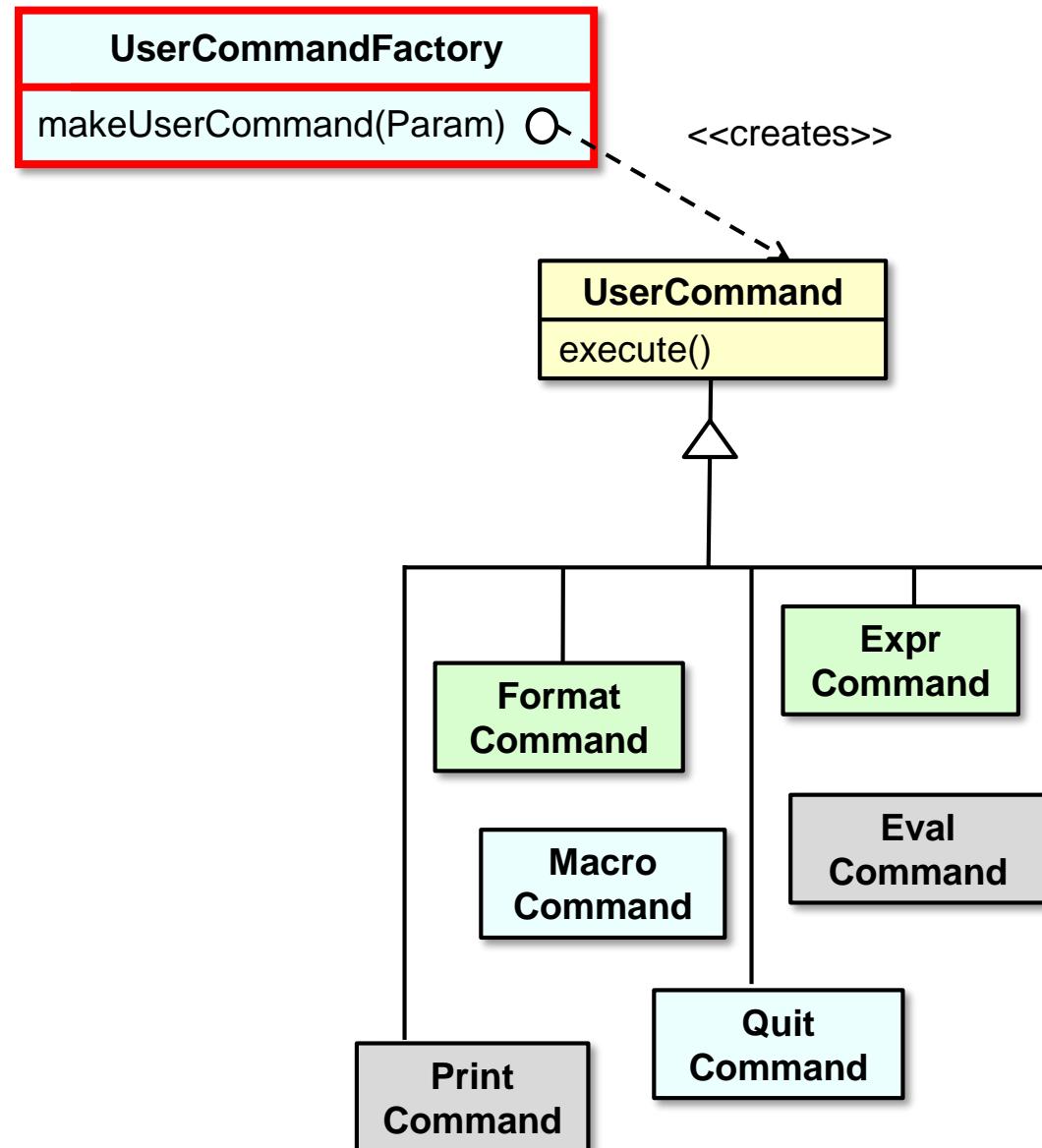
Applicability

- When a class cannot anticipate the objects it must create
- A class wants its subclasses to specify the objects it creates
 - This approach is optional



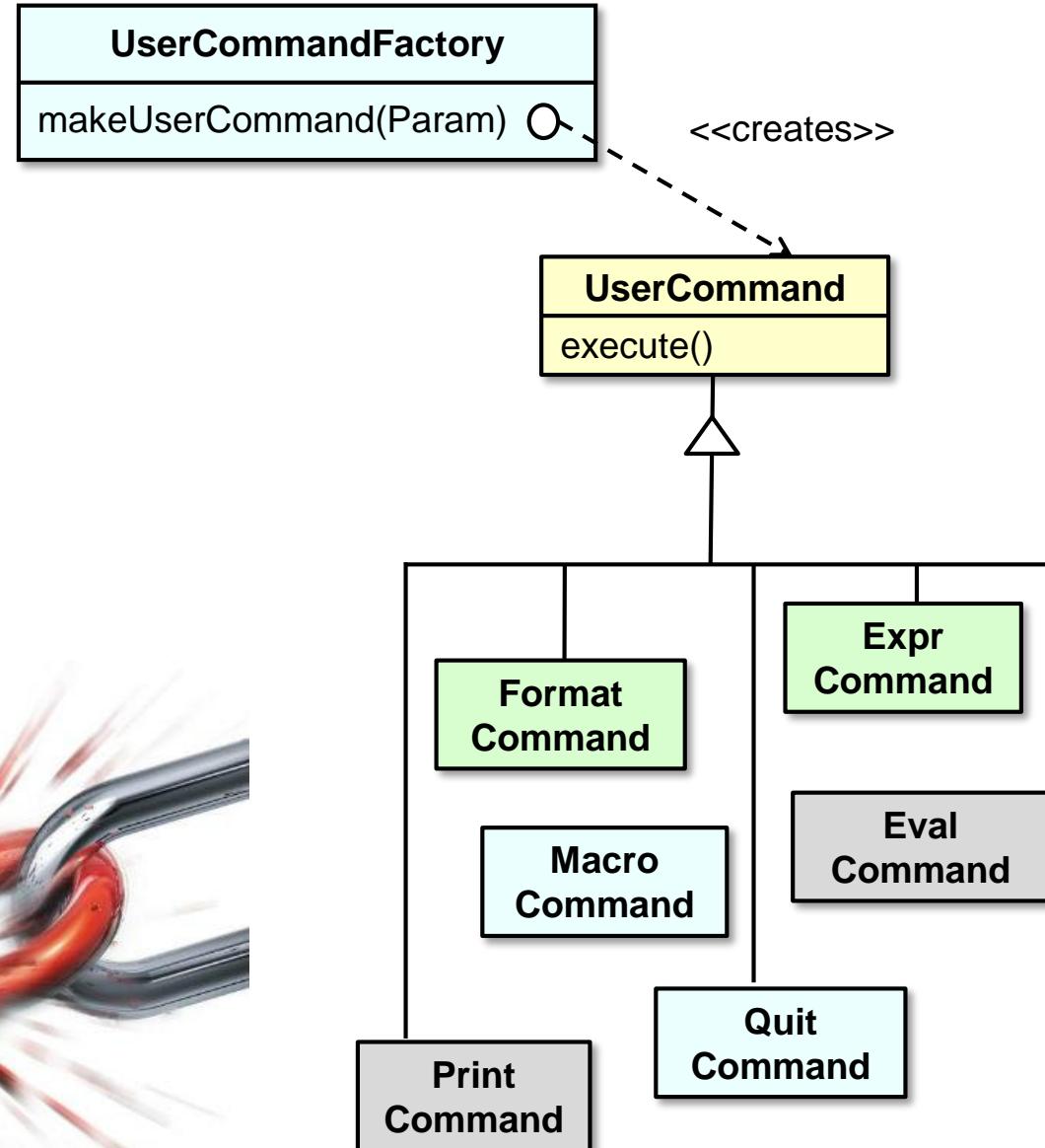
Applicability

- When a class cannot anticipate the objects it must create
- A class wants its subclasses to specify the objects it creates
 - This approach is optional
 - An alternative involves passing a parameter to the factory method

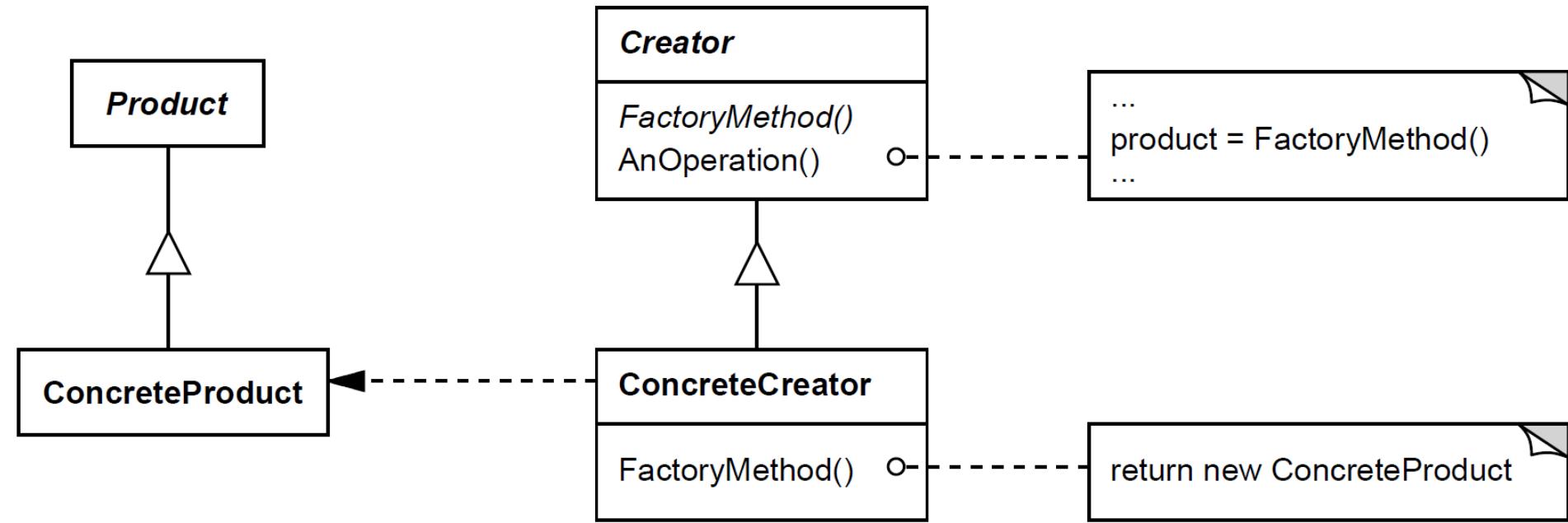


Applicability

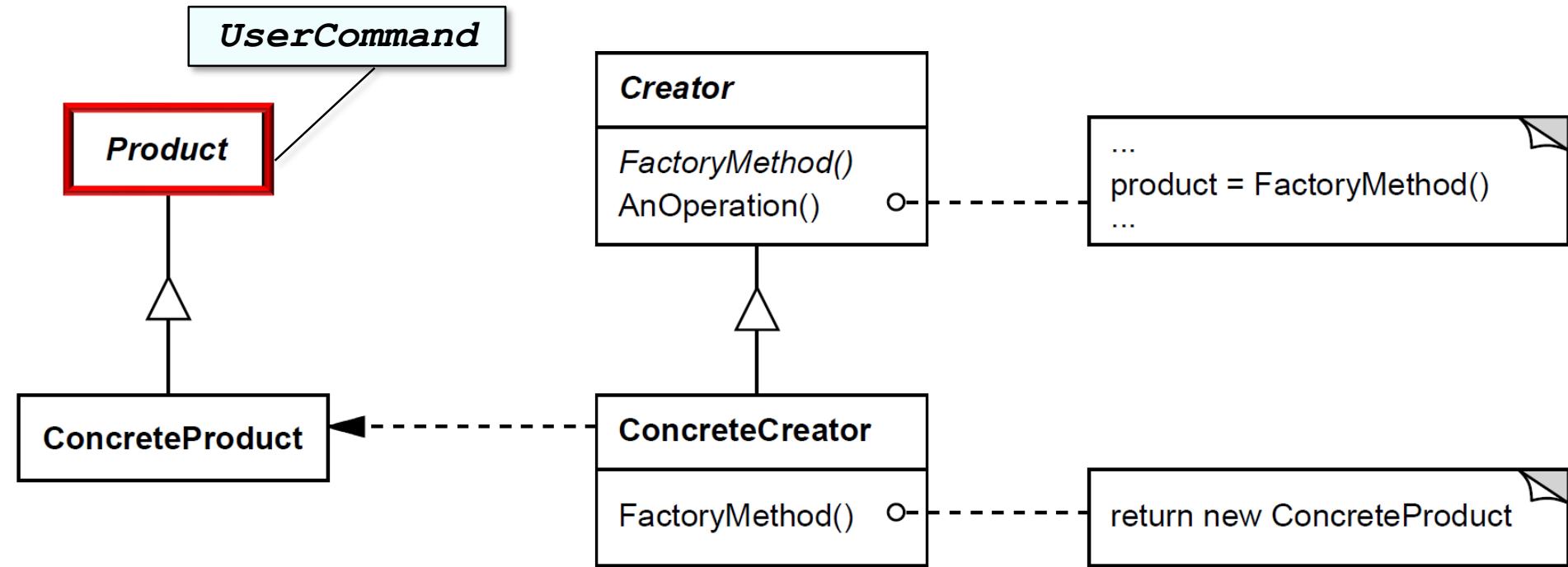
- When a class cannot anticipate the objects it must create
- A class wants its subclasses to specify the objects it creates
- Or there's a need to decouple object creation from its subsequent use



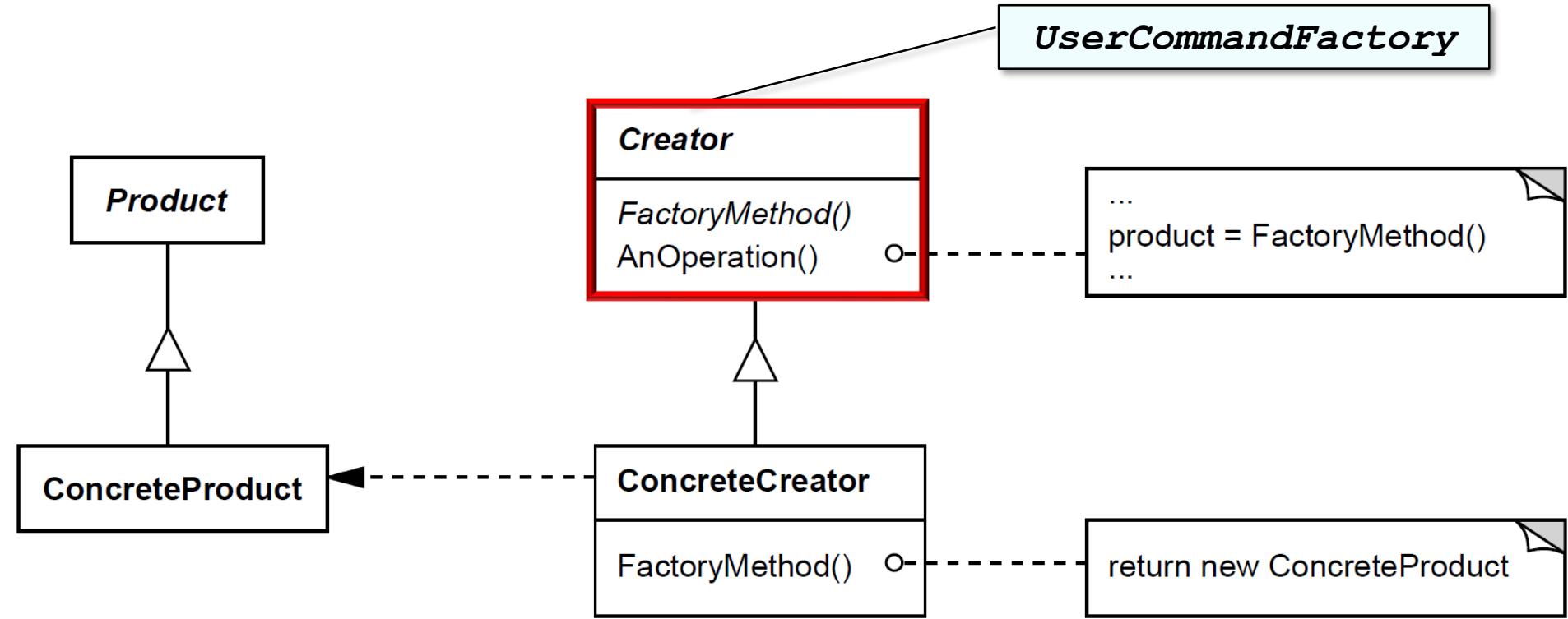
Structure & Participants



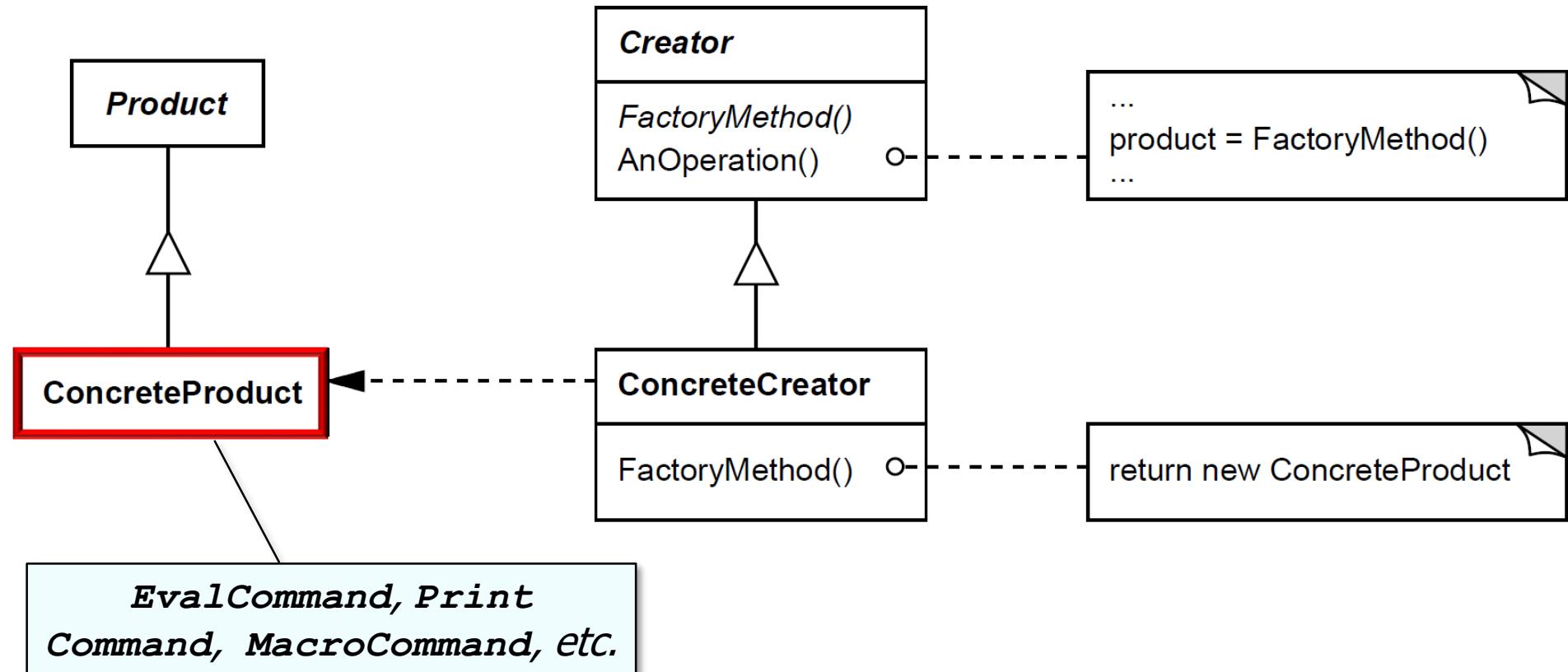
Structure & Participants



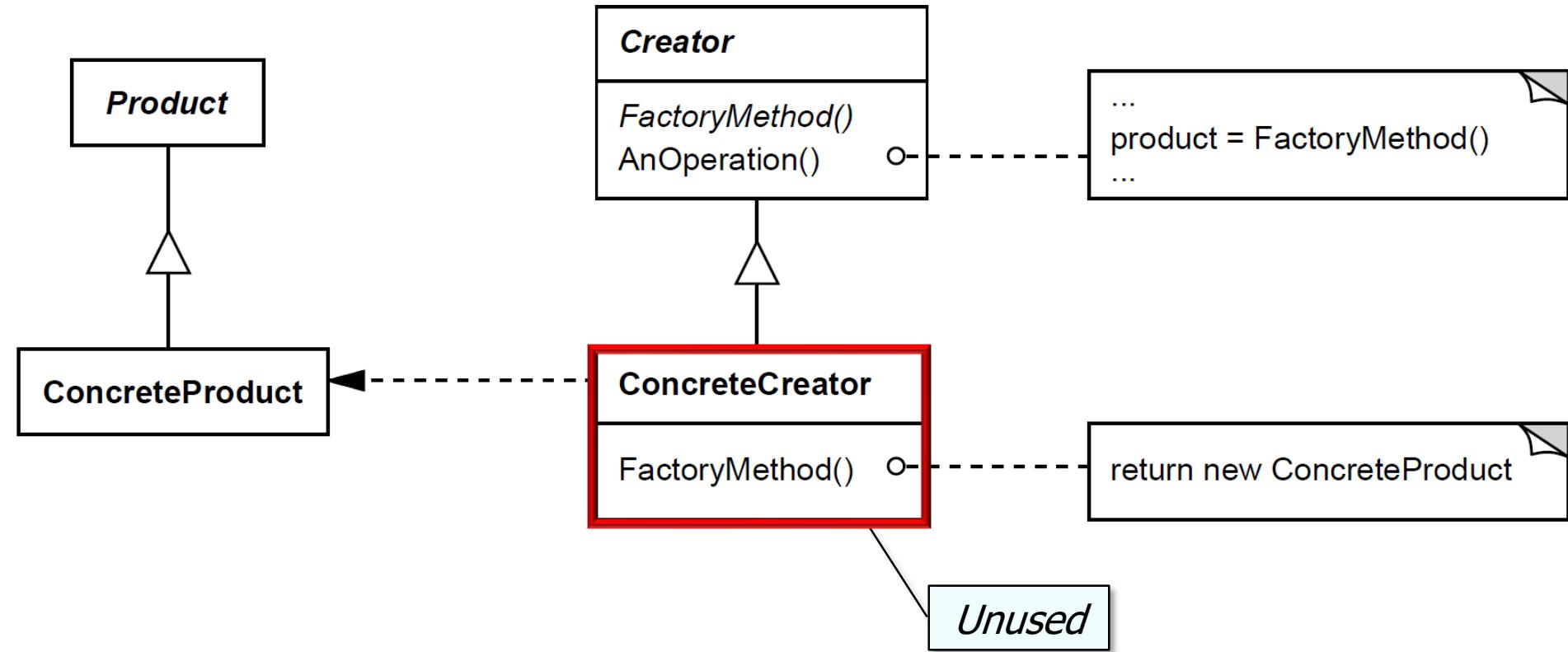
Structure & Participants



Structure & Participants



Structure & Participants



Our app passes a string to the factory method rather than using subclassing

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {  
  
    private interface UserCommandFactoryCommand  
    { UserCommand execute(String param) ; }  
  
    private HashMap<String,UserCommandFactoryCommand> mCommandMap =  
    new HashMap<>();  
  
    UserCommandFactory(final TreeContext treeOps) {  
        mCommandMap.put("format", new UserCommandFactoryCommand() {  
            public UserCommand execute(String param)  
            { return new FormatCommand(treeOps, param) ; }  
        });  
        ...  
    }  
}
```

See [ExpressionTree/CommandLine/src/expressiontree/commands](#)

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory { ← We first apply Command to  
    private interface UserCommandFactoryCommand  
    { UserCommand execute(String param); }  
  
    private HashMap<String,UserCommandFactoryCommand> mCommandMap =  
        new HashMap<>();  
  
    UserCommandFactory(final TreeContext treeOps) {  
        mCommandMap.put("format", new UserCommandFactoryCommand() {  
            public UserCommand execute(String param)  
            { return new FormatCommand(treeOps, param); }  
        });  
        ...  
    }  
}
```

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {  
    private interface UserCommandFactoryCommand  
    { UserCommand execute(String param) ; }  
  
    private HashMap<String,UserCommandFactoryCommand> mCommandMap =  
    new HashMap<>();  
  
    UserCommandFactory(final TreeContext treeOps) {  
        mCommandMap.put("format", new UserCommandFactoryCommand() {  
            public UserCommand execute(String param)  
            { return new FormatCommand(treeOps, param) ; }  
        });  
        ...  
    }  
}
```



Command interface

UserCommandFactoryCommand is a Java 8 “functional interface”

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {  
  
    private interface UserCommandFactoryCommand  
    { UserCommand execute(String param) ; }  
  
    private HashMap<String,UserCommandFactoryCommand> mCommandMap =  
        new HashMap<>();  
  
    Map strings to factory commands that create UserCommand objects   
  
    UserCommandFactory(final TreeContext treeOps) {  
        mCommandMap.put("format", new UserCommandFactoryCommand () {  
            public UserCommand execute(String param)  
            { return new FormatCommand(treeOps, param) ; }  
        });  
        ...  
    }  
}
```

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {  
  
    private interface UserCommandFactoryCommand  
    { UserCommand execute(String param) ; }  
  
    private HashMap<String,UserCommandFactoryCommand> mCommandMap =  
    new HashMap<>();  
  
    UserCommandFactory(final TreeContext treeOps) {  
        mCommandMap.put("format", new UserCommandFactoryCommand () {  
            public UserCommand execute(String param)  
            { return new FormatCommand(treeOps, param) ; }  
        });  
        ...  
    }  
}
```



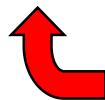
An anonymous inner class defines a factory command that creates a FormatCommand



Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {  
  
    private interface UserCommandFactoryCommand  
    { UserCommand execute(String param) ; }  
  
    private HashMap<String,UserCommandFactoryCommand> mCommandMap =  
    new HashMap<>();  
  
    UserCommandFactory(final TreeContext treeOps) {  
        mCommandMap.put("format", param ->  
            new FormatCommand(treeOps, param));  
        ...  
    }  
}
```



Java 8 lambda that creates a FormatCommand

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {
```

The factory method

```
    public UserCommand makeUserCommand (String inputString) {  
        String commandRequest = ... /* get command from inputString */  
        String parameters = ... /* get parameters from inputString */
```

```
        UserCommandFactoryCommand command =  
            mCommandMap.get(commandRequest);  
  
        if (command != null)  
            return command.execute(parameters);  
        else  
            return new QuitCommand(mTreeContext);  
    ...
```

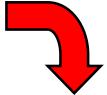
The factory method uses a map to find/execute the command that makes a command

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {
```

```
    public UserCommand makeUserCommand (String inputString) {  
        String commandRequest = ... /* get command from inputString */  
        String parameters = ... /* get parameters from inputString */
```

Try to find pre-allocated factory command 
UserCommandFactoryCommand command =
 mCommandMap.get(commandRequest);

```
    if (command != null)  
        return command.execute(parameters);  
    else  
        return new QuitCommand(mTreeContext);  
    ...
```

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {
```

```
    public UserCommand makeUserCommand (String inputString) {  
        String commandRequest = ... /* get command from inputString */  
        String parameters = ... /* get parameters from inputString */
```

```
        UserCommandFactoryCommand command =  
            mCommandMap.get(commandRequest);
```

```
        if (command != null) {  
            return command.execute(parameters);  
        } else  
            return new QuitCommand(mTreeContext);  
        ...
```



If found execute it to make a command

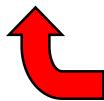
return command.execute(parameters);

new QuitCommand(mTreeContext);

Factory Method example in Java

- UserCommandFactory creates UserCommands based on user input

```
public class UserCommandFactory {  
  
    public UserCommand makeUserCommand (String inputString) {  
        String commandRequest = ... /* get command from inputString */  
        String parameters = ... /* get parameters from inputString */  
  
        UserCommandFactoryCommand command =  
            mCommandMap.get(commandRequest);  
  
        if (command != null)  
            return command.execute(parameters);  
        else  
            return new QuitCommand(mTreeContext);  
    ...  
}
```



Otherwise, user gave an unsupported request, so quit

Consequences

+ Decoupling

- Client are more flexible since they needn't specify the class name of the concrete class & the details of its creation

Instead of

```
UserCommand command =  
    new PrintCommand();
```

Use

```
UserCommand command  
= userCommandfactory.  
    makeUserCommand  
    ("print"));
```

where `userCommandFactory` is an instance of `UserCommandFactory`

Consequences

+ Decoupling

- Client are more flexible since they needn't specify the class name of the concrete class & the details of its creation

Instead of

UserCommand command =
new PrintCommand();



Use

```
UserCommand command  
= userCommandfactory.  
makeUserCommand  
("print");
```

where **userCommandFactory** is an instance of **UserCommandFactory**

Consequences

+ Decoupling

- Client are more flexible since they needn't specify the class name of the concrete class & the details of its creation

Instead of

```
UserCommand command =  
    new PrintCommand();
```

Use

No lexical dependency
on any concrete class

```
UserCommand command  
= userCommandfactory.  
    makeUserCommand  
    ("print");
```

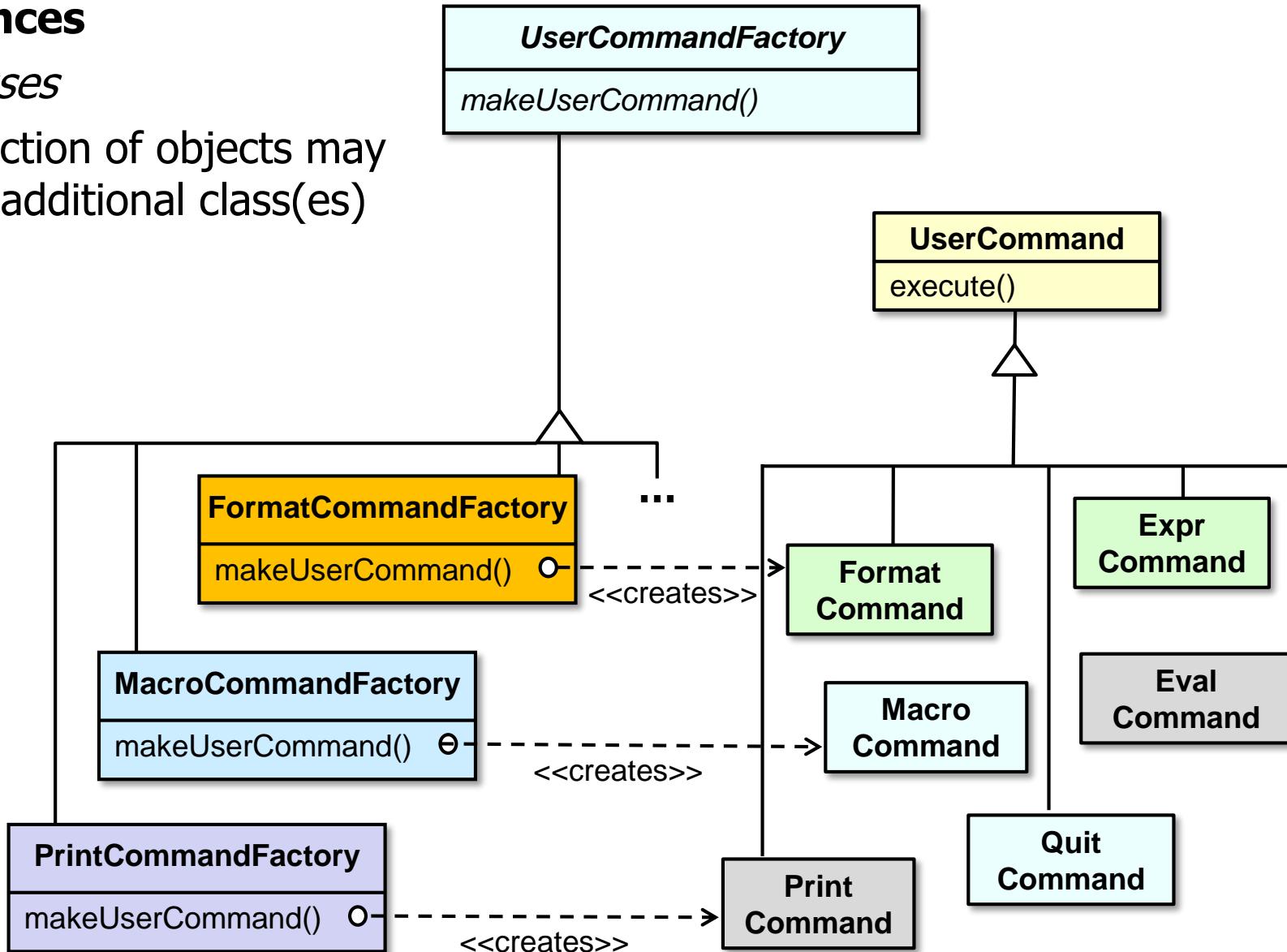
where `userCommandFactory` is an instance of `UserCommandFactory`



Consequences

– *More classes*

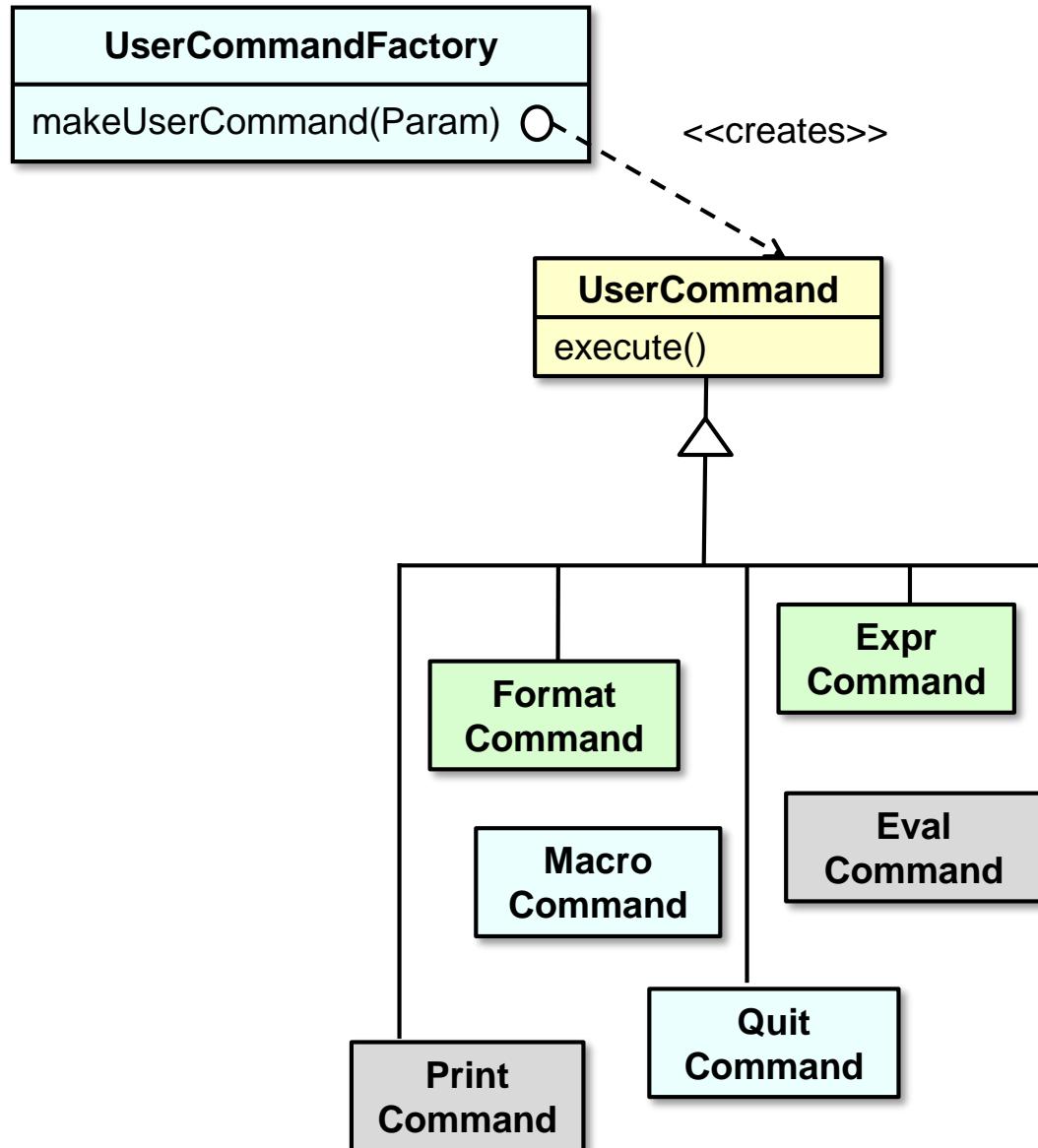
- Construction of objects may require additional class(es)



Consequences

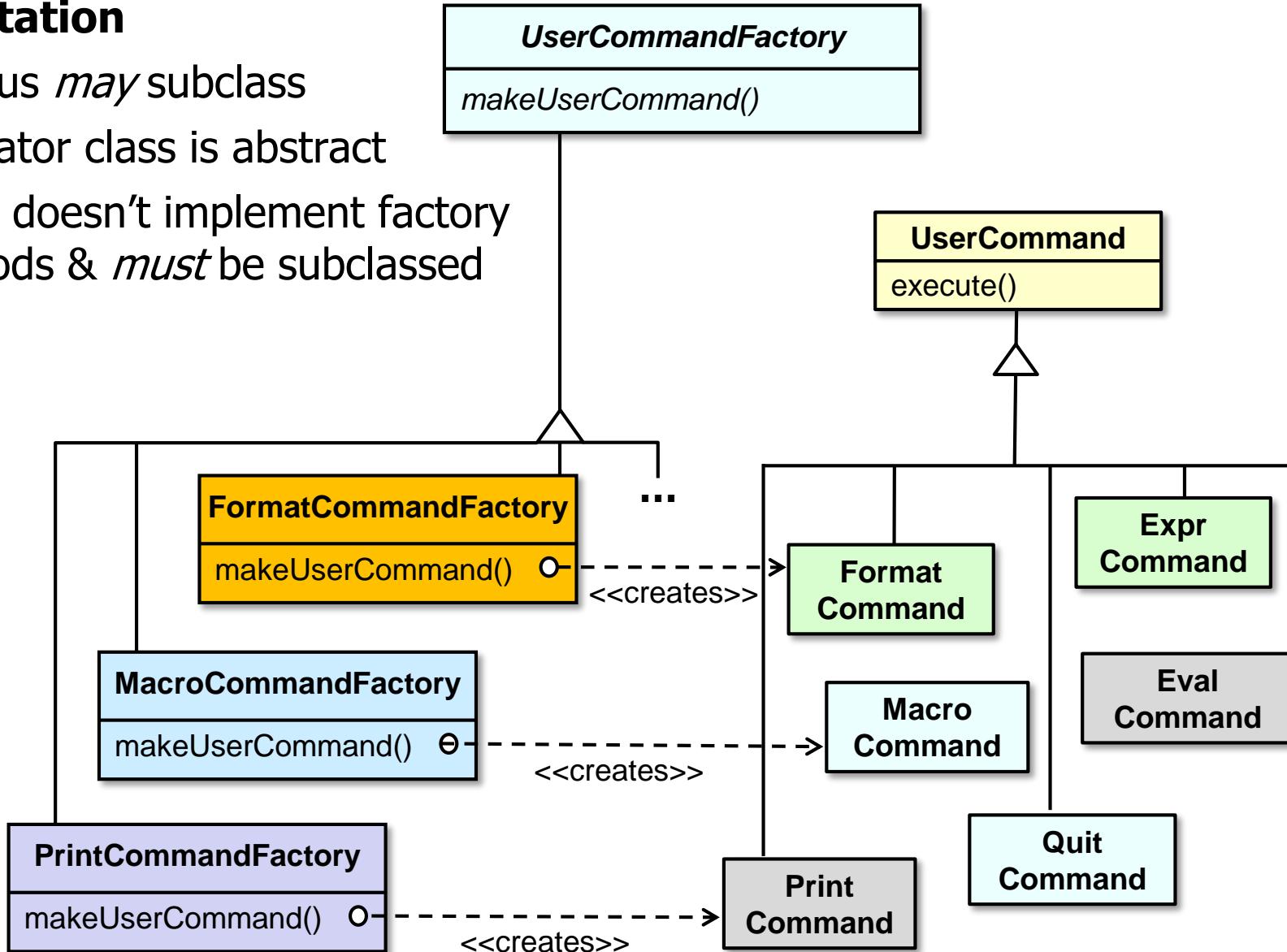
– *More classes*

- Construction of objects may require additional class(es)
- An alternative is to pass a param to Creator super class factory method



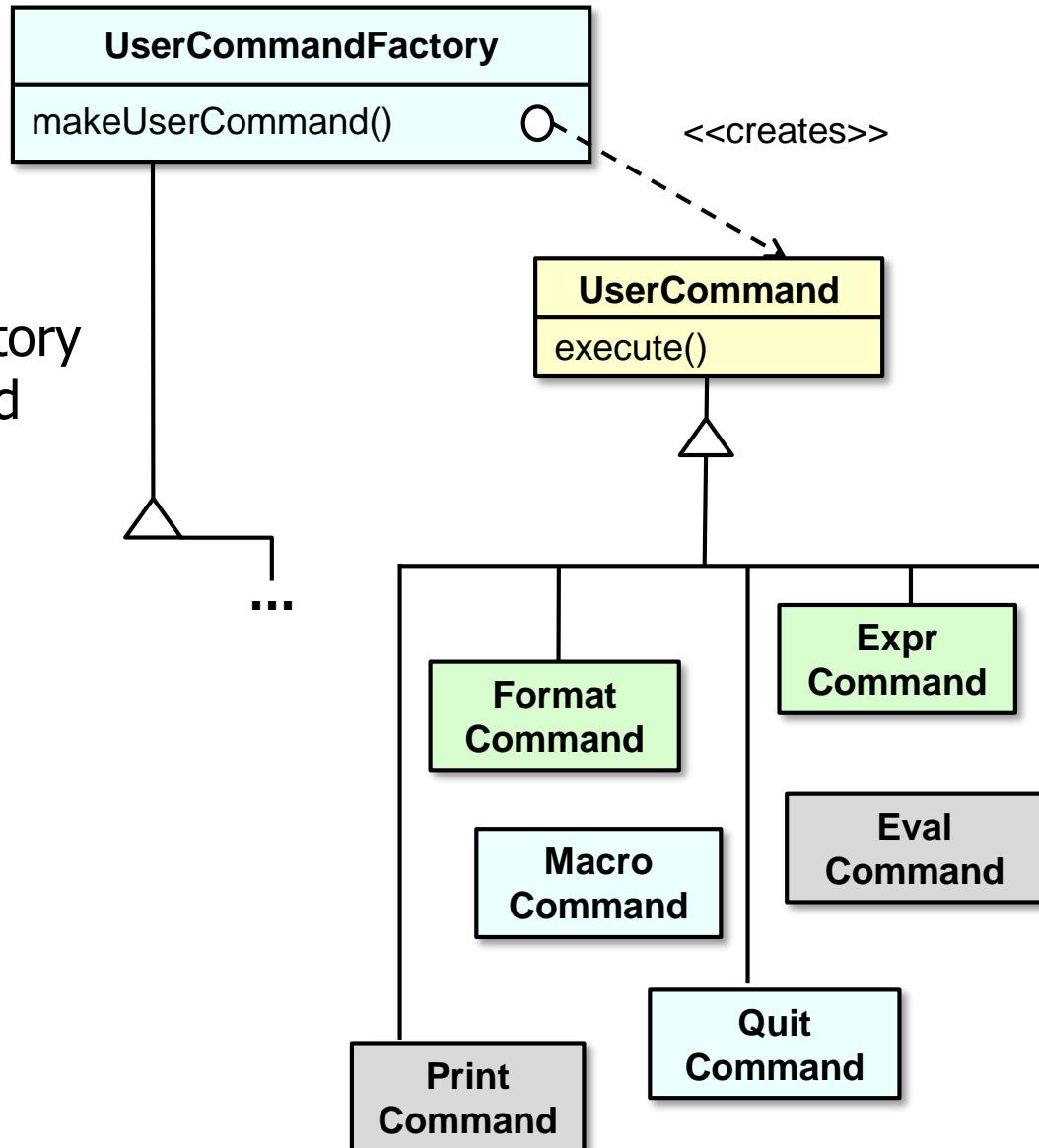
Implementation

- *Must* versus *may* subclass
 - The creator class is abstract
 - i.e., it doesn't implement factory methods & *must* be subclassed



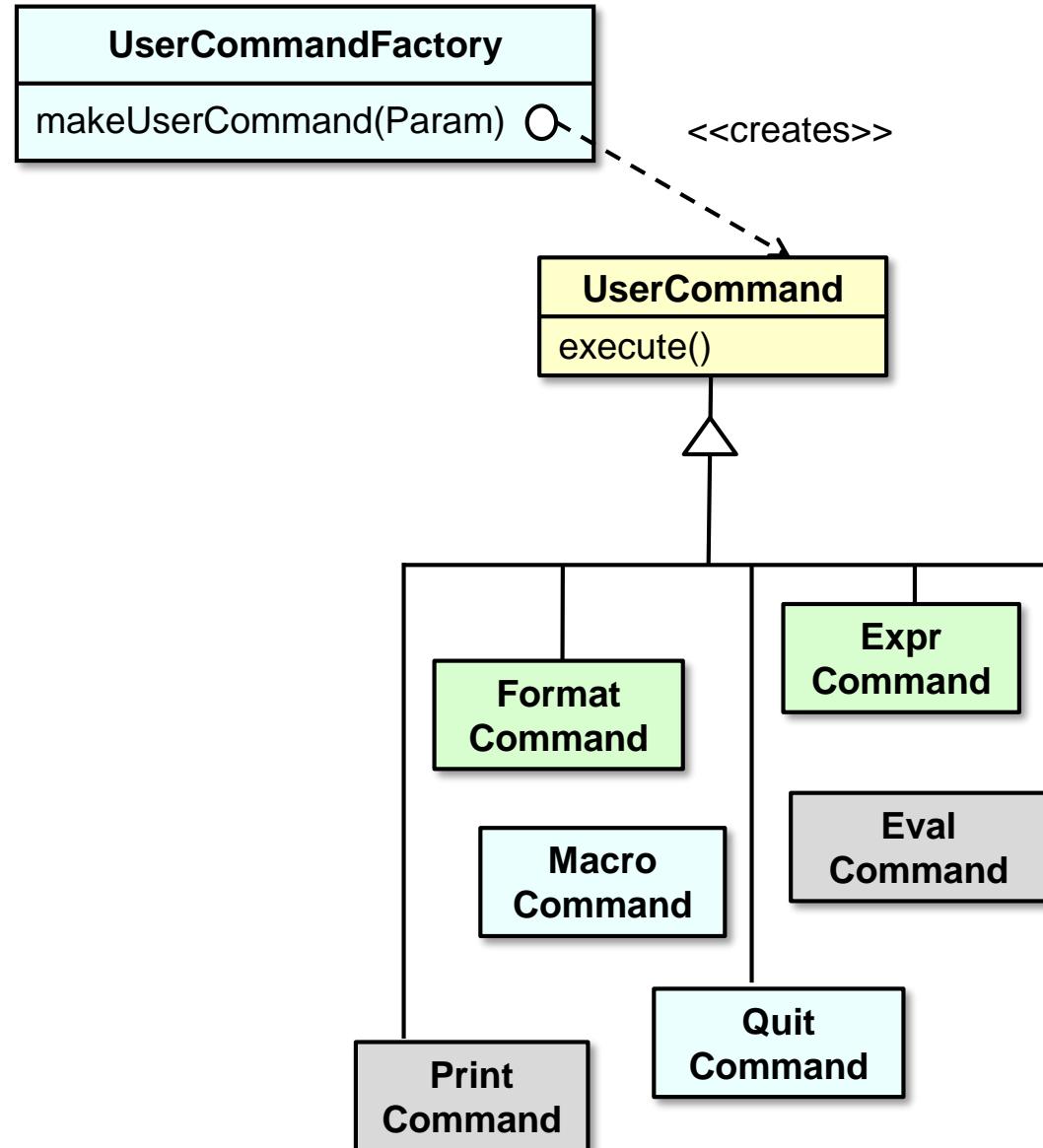
Implementation

- *Must* versus *may* subclass
 - The creator class is abstract
 - The creator class is concrete
 - i.e., it provides a default factory method & *may* be subclassed



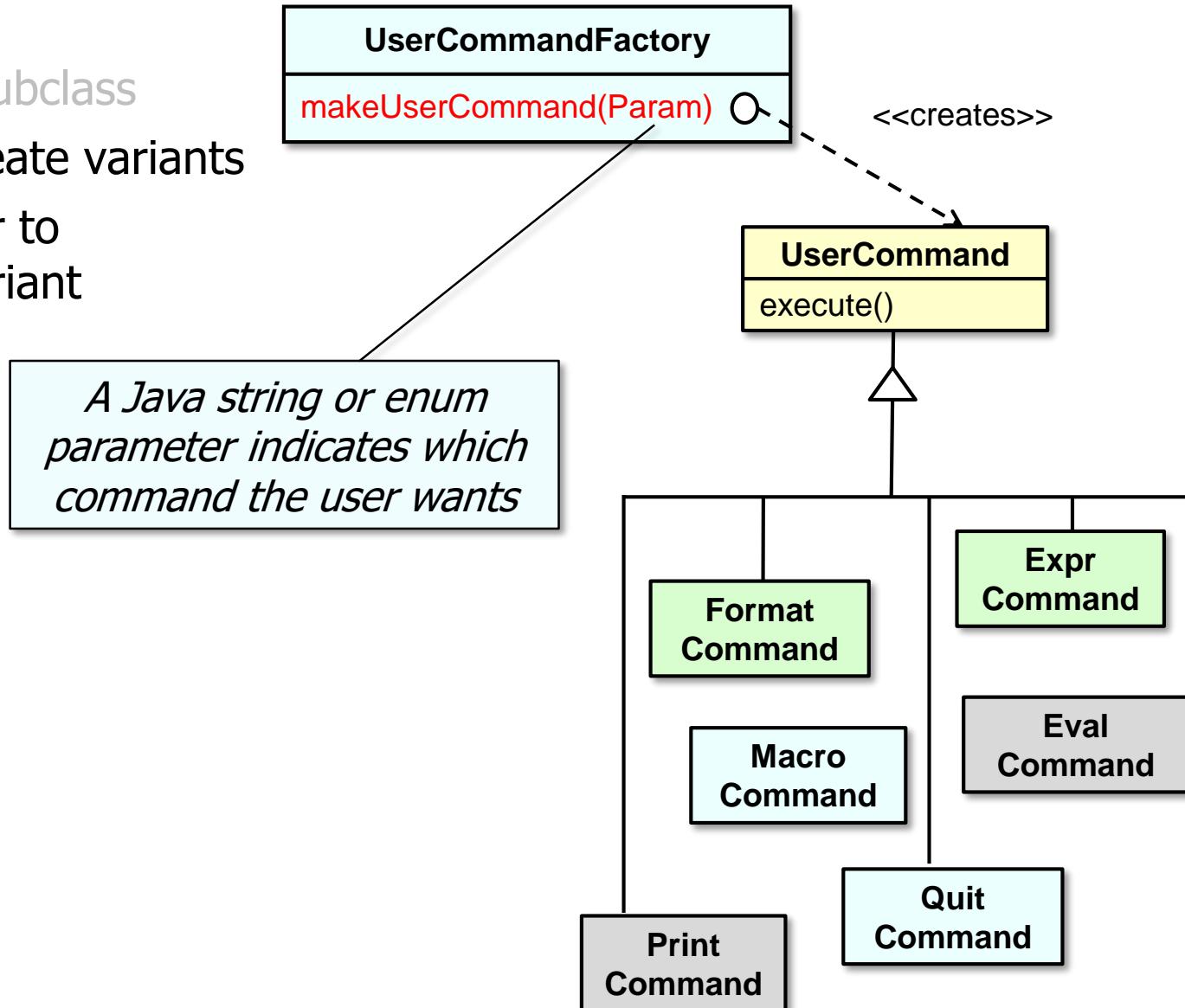
Implementation

- *Must* versus *may* subclass
- Factory method create variants
 - Pass a parameter to designate the variant



Implementation

- *Must* versus *may* subclass
- Factory method create variants
 - Pass a parameter to designate the variant



A string is more flexible, whereas an enum is more type-safe

Implementation

- *Must* versus *may* subclass
- Factory method create variants
- Constructor references in Java 8 may reduce the tedium of creating Product subclasses

```
class ShapeFactory {  
    Map<String, Supplier<Shape>> map =  
        new HashMap<>() {{  
            put("CIRCLE", Circle::new);  
            put("RECTANGLE",  
                Rectangle::new);  
            ...  
        }};  
  
    public Shape getShape(String shape) {  
        Supplier<Shape> shape = map  
            .get(shapeType.toUpperCase());  
        if (shape != null)  
            return shape.get();  
        throw new IllegalArgumentException  
            ("No such shape " +  
            shape.toUpperCase());  
    }  
}
```

Implementation

- Must versus may subclass
- Factory method create variants
- Constructor references in Java 8 may reduce the tedium of creating Product subclasses

```
class ShapeFactory {  
    Map<String, Supplier<Shape>> map =  
        new HashMap<>() {{  
            put("CIRCLE", Circle::new);  
            put("RECTANGLE",  
                Rectangle::new);  
            ...  
        }};
```

Constructor references can be used to create desired shapes

```
public getShape(String shape) {  
    Supplier<Shape> shape = map  
        .get(shapeType.toUpperCase());  
    if (shape != null)  
        return shape.get();  
    throw new IllegalArgumentException  
        ("No such shape " +  
        shape.toUpperCase());  
}
```

Implementation

- *Must* versus *may* subclass
- Factory method create variants
- Constructor references in Java 8 may reduce the tedium of creating Product subclasses

```
class ShapeFactory {  
    Map<String, Supplier<Shape>> map =  
        new HashMap<>() {{  
            put("CIRCLE", Circle::new);  
            put("RECTANGLE",  
                Rectangle::new);  
            ...  
        }};  
  
    public getShape(String shape) {  
        Supplier<Shape> shape = map  
            .get(shapeType.toUpperCase());  
        if (shape != null)  
            return shape.get();  
        throw new IllegalArgumentException  
            ("No such shape " +  
            shape.toUpperCase());  
    }  
}
```

Create the requested
Shape subclass

Implementation

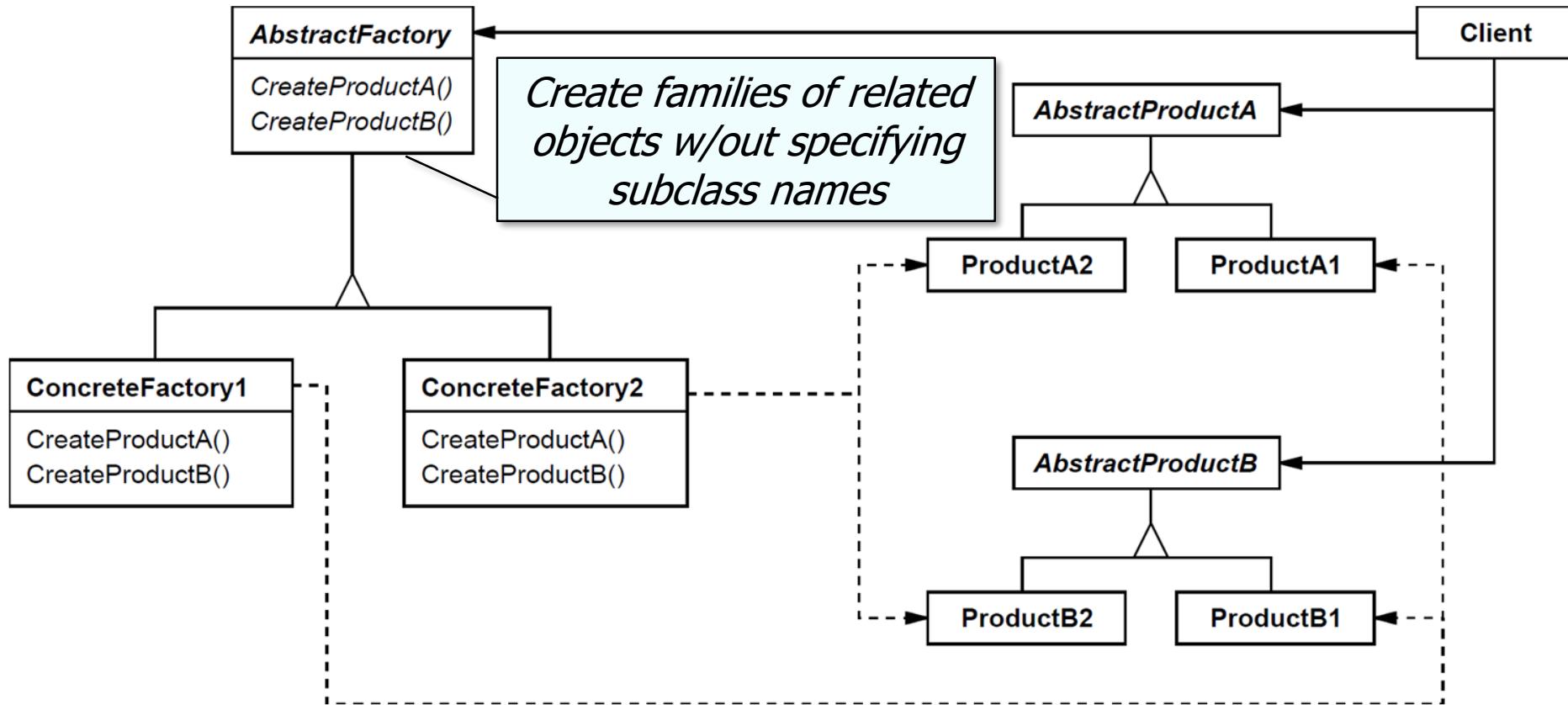
- Must versus may subclass
- Factory method create variants
- Constructor references in Java 8 may reduce the tedium of creating Product subclasses

Doesn't scale if `getShape()` takes multiple arguments to pass to `Shape` constructors

```
class ShapeFactory {  
    Map<String, Supplier<Shape>> map =  
        new HashMap<>() {{  
            put("CIRCLE", Circle::new);  
            put("RECTANGLE",  
                Rectangle::new);  
            ...  
        }};  
  
    public Shape getShape(String shape) {  
        Supplier<Shape> shape = map  
            .get(shapeType.toUpperCase());  
        if (shape != null)  
            return shape.get();  
        throw new IllegalArgumentException  
            ("No such shape " +  
            shape.toUpperCase());  
    }  
}
```

Implementation

- *Must* versus *may* subclass
- Factory method create variants
- Apply *Abstract Factory* if many semantically-consistent factory methods needed



See en.wikipedia.org/wiki/Abstract_factory_pattern

Known Uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- BREW feature phone frameworks
- The ACE ORB (TAO)
- **URLStreamHandlerFactory** in Java

Interface URLStreamHandlerFactory

```
public interface URLStreamHandlerFactory
```

This interface defines a factory for URL stream protocol handlers.

It is used by the URL class to create a URLStreamHandler for a specific protocol.

Since:

JDK1.0

See Also:

[URL](#), [URLStreamHandler](#)

Method Summary

[All Methods](#) [Instance Methods](#) [Abstract Methods](#)

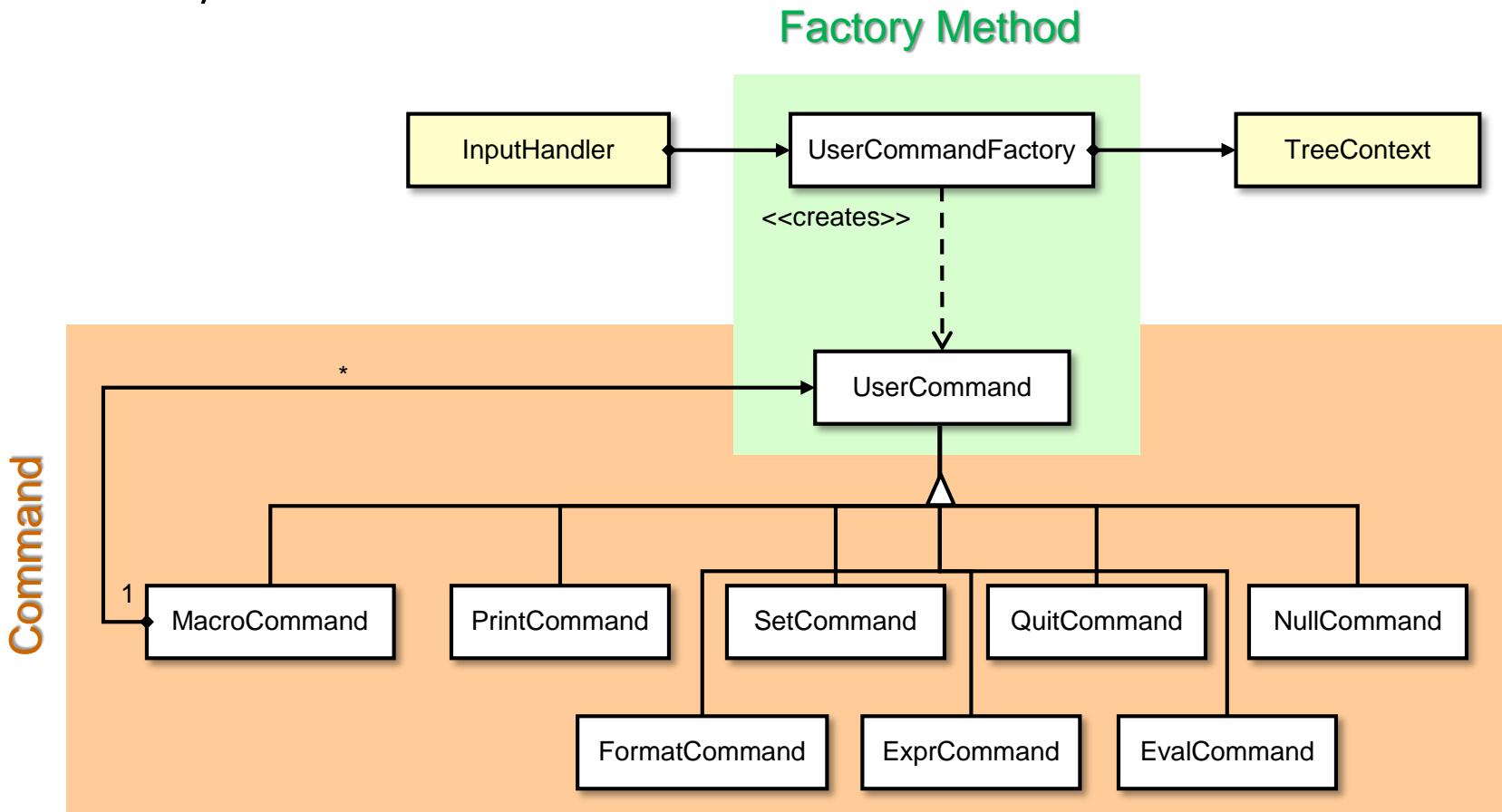
Modifier and Type **Method and Description**

`URLStreamHandler createURLStreamHandler(String protocol)`

Creates a new URLStreamHandler instance with the specified protocol.

Summary of the Factory Method Pattern

- *Factory Method* enables extensible creation of variabilities, such as iterators, commands, & visitors



Factory Method decouples the creation of objects from their subsequent use

End of the
Factory Method Pattern

Overview of Pattern Concepts

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt



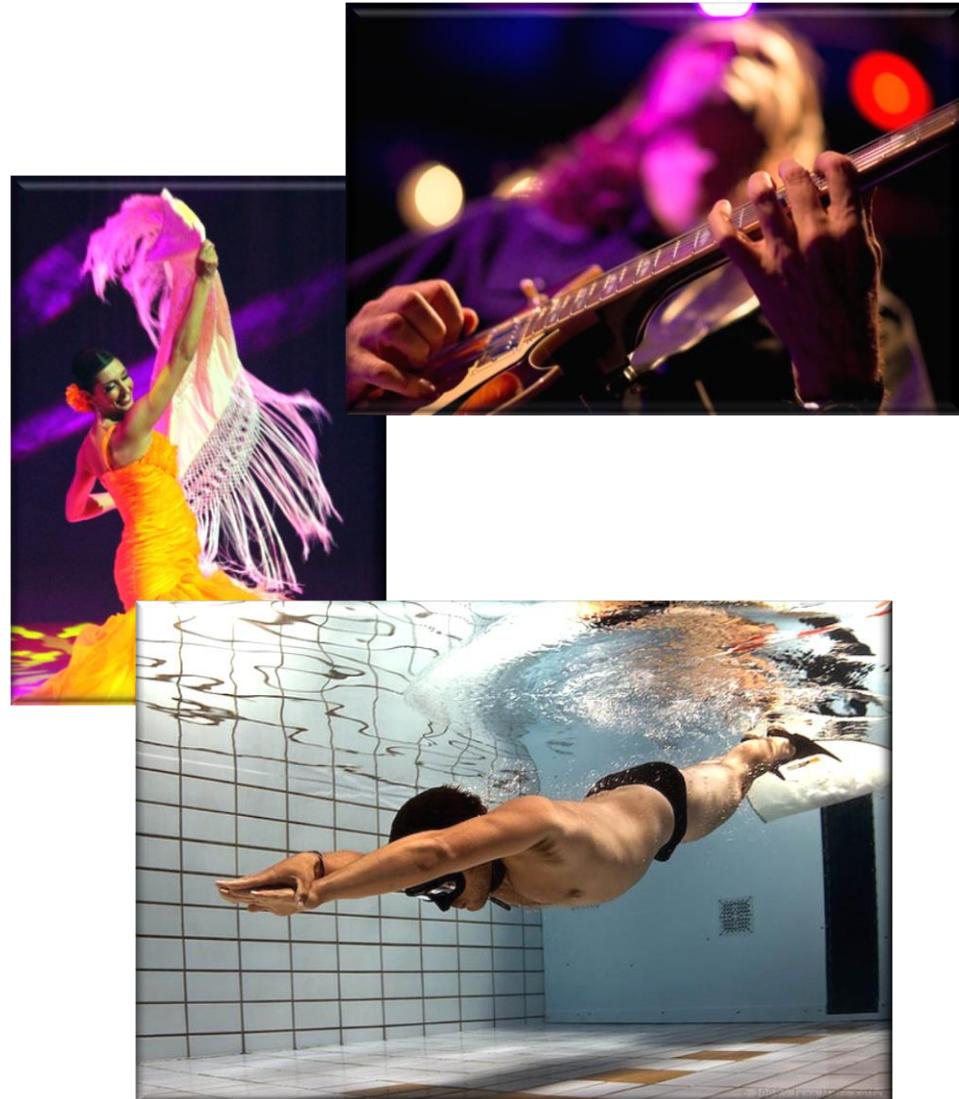
Professor of Computer Science
Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA



Lesson Intro

- Experts perform differently than beginners
 - Unlike novices, professional athletes, musicians, & dancers move fluidly & effortlessly, without focusing on each individual movement



Lesson Intro

- When watching experts perform it's easy to forget how much effort they've put into reaching high levels of achievement



Lesson Intro

- Continuous repetition & practice are crucial to their success



"Explosively entertaining... *Outliers* is riveting science, self-help, and entertainment, all in one book."

— ENTERTAINMENT WEEKLY

#1 National Bestseller

Outliers



THE STORY OF SUCCESS

MALCOLM
GLADWELL

Author of *The Tipping Point* and *Blink*

Lesson Intro

- Mentoring from other experts is also essential to achieving mastery



Lesson Intro

At the heart of all these activities is knowledge & mastery of *patterns*



"Explosively entertaining... *Outliers* is riveting science, self-help, and entertainment, all in one book."
— ENTERTAINMENT WEEKLY

#1 National Bestseller

Outliers



THE STORY OF SUCCESS

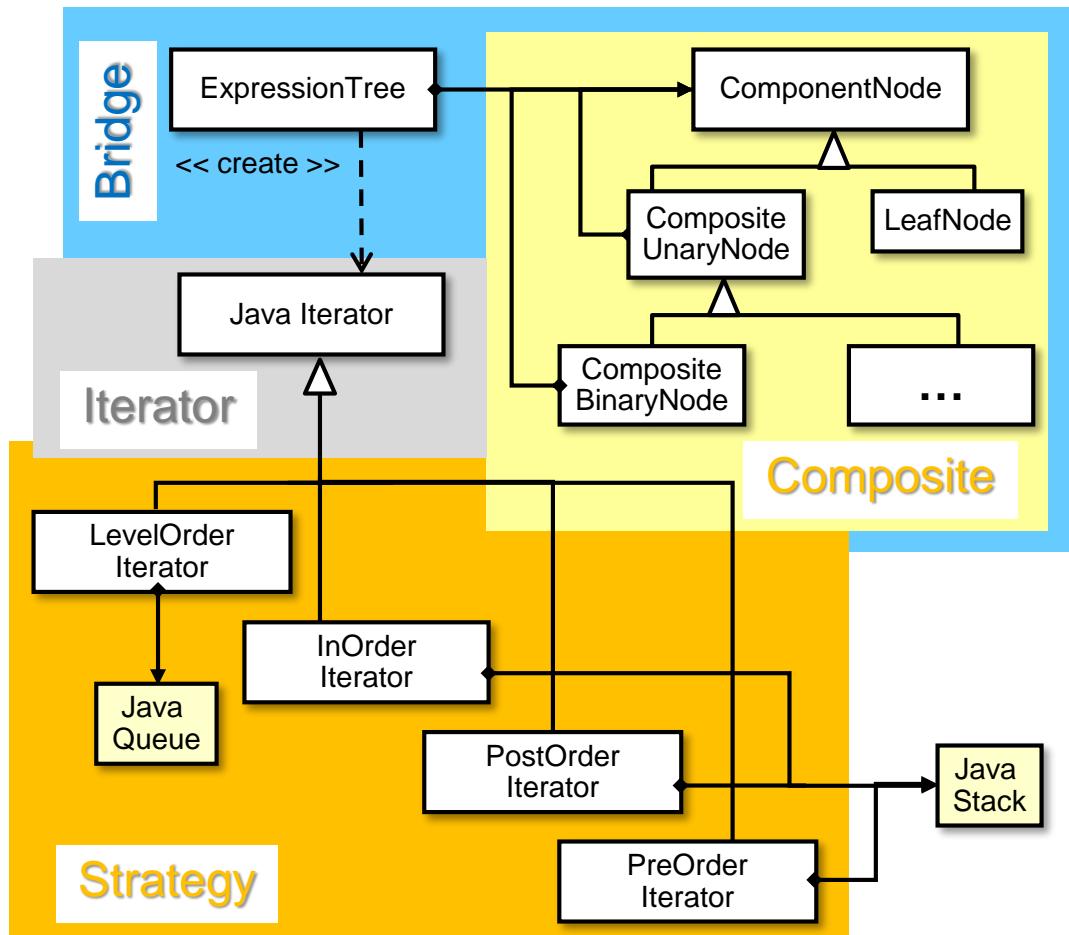
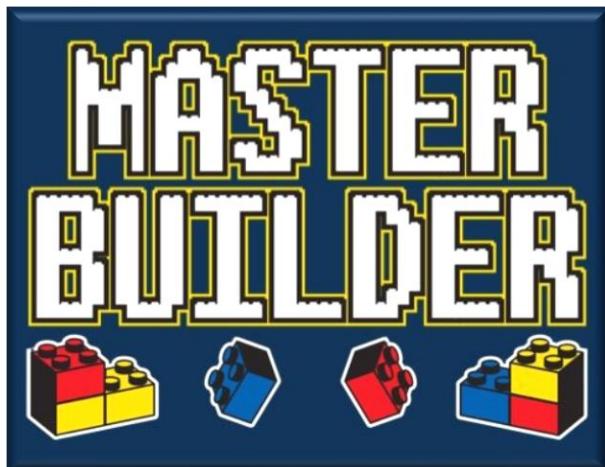
MALCOLM
GLADWELL

Author of *The Tipping Point* and *Blink*

Learning Objectives

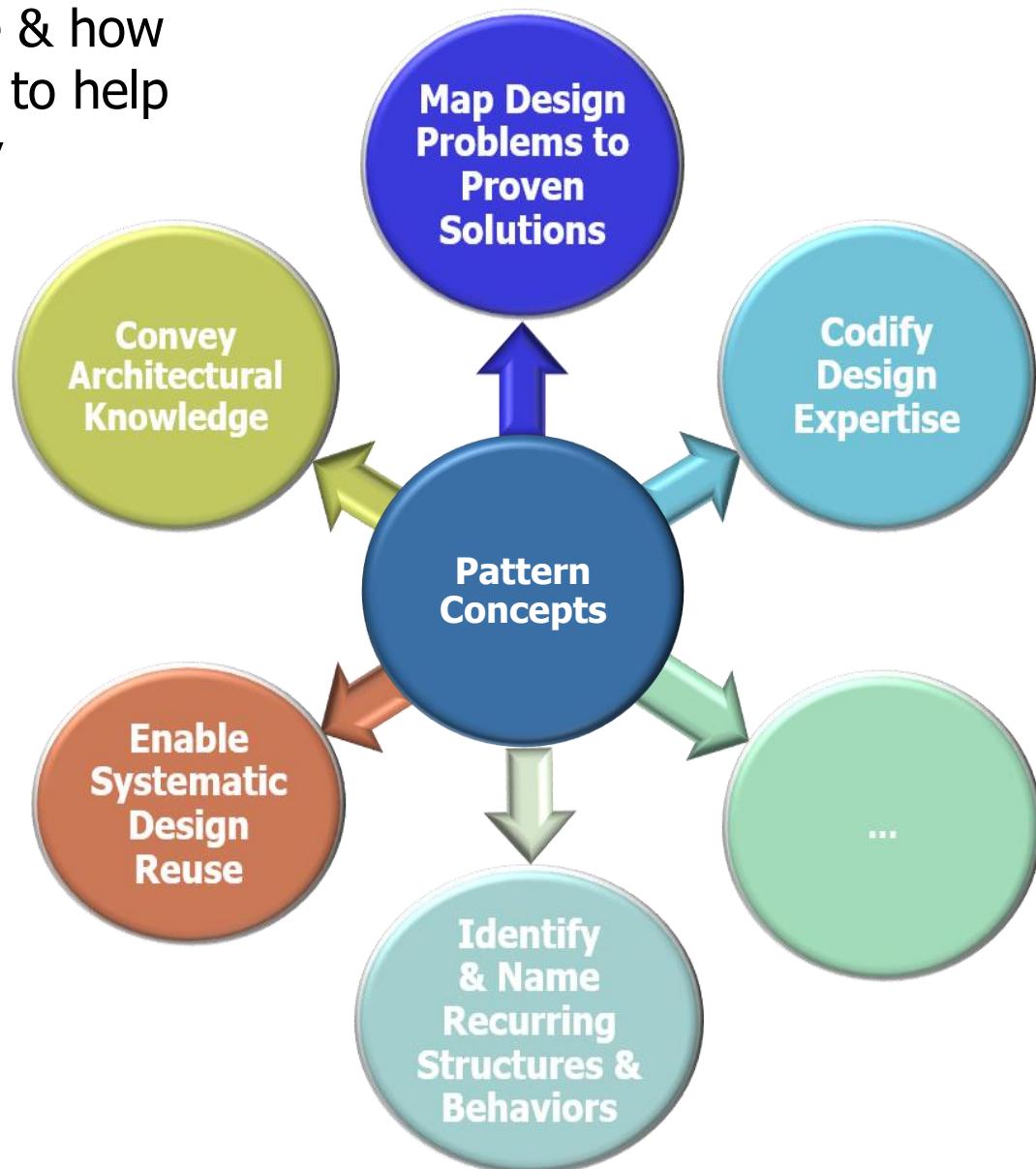
Learning Objectives

- Recognize the importance of design experience in the quest to become a master software developer



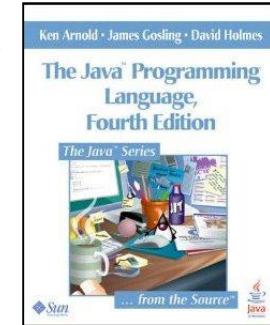
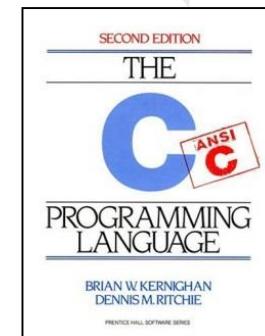
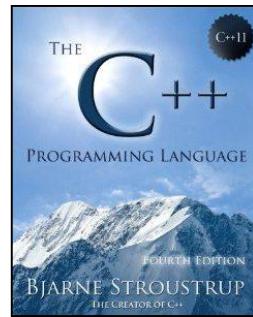
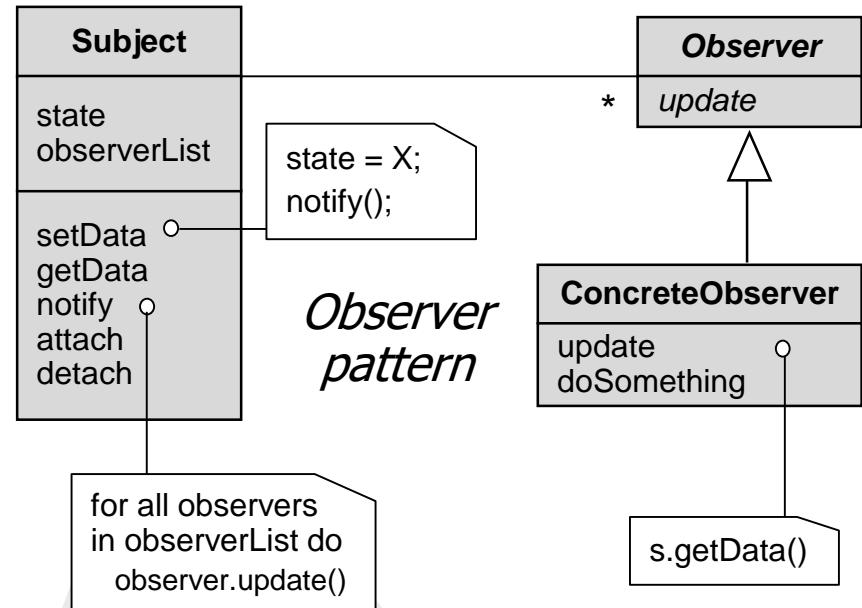
Learning Objectives

- Understand what patterns are & how they codify design experience to help improve quality & productivity



Learning Objectives

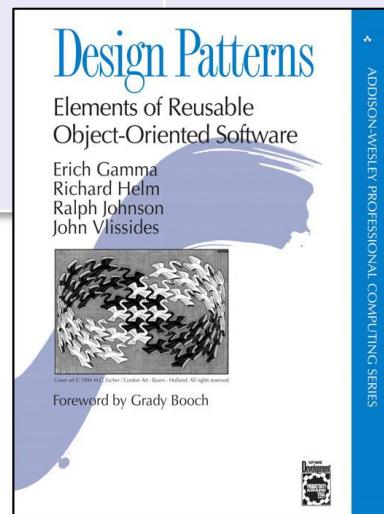
- Identify common characteristics of patterns & pattern descriptions



Learning Objectives

- Know the history of the “Gang of Four” (Gof) book & its patterns

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



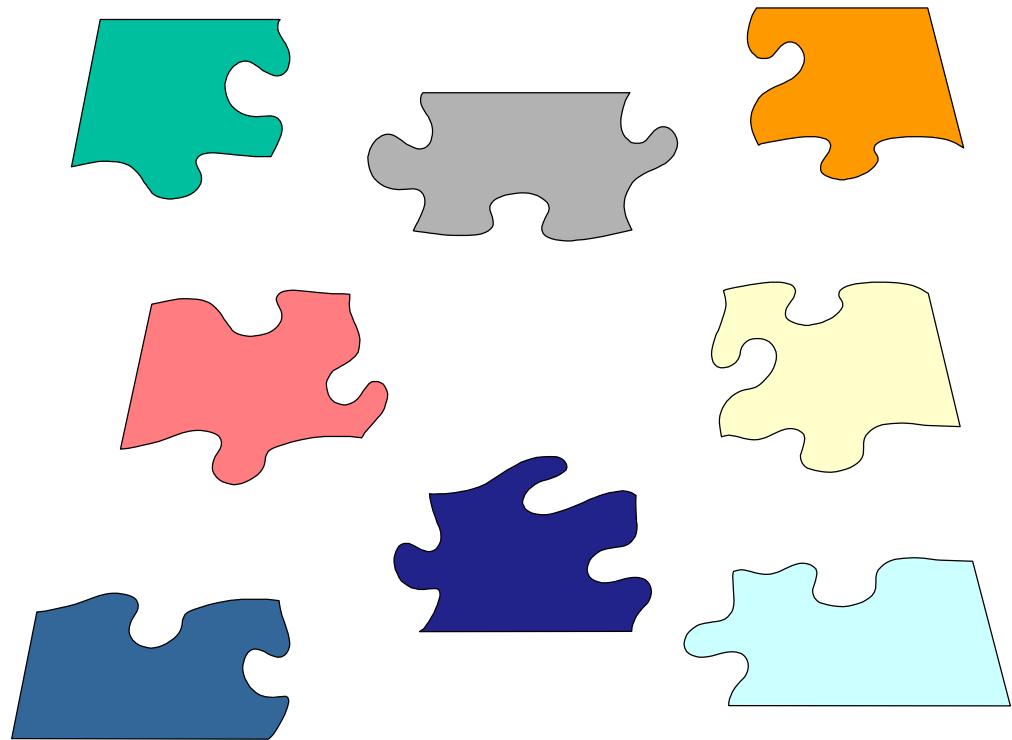
Learning Objectives

- Know the key relationships between patterns



Learning Objectives

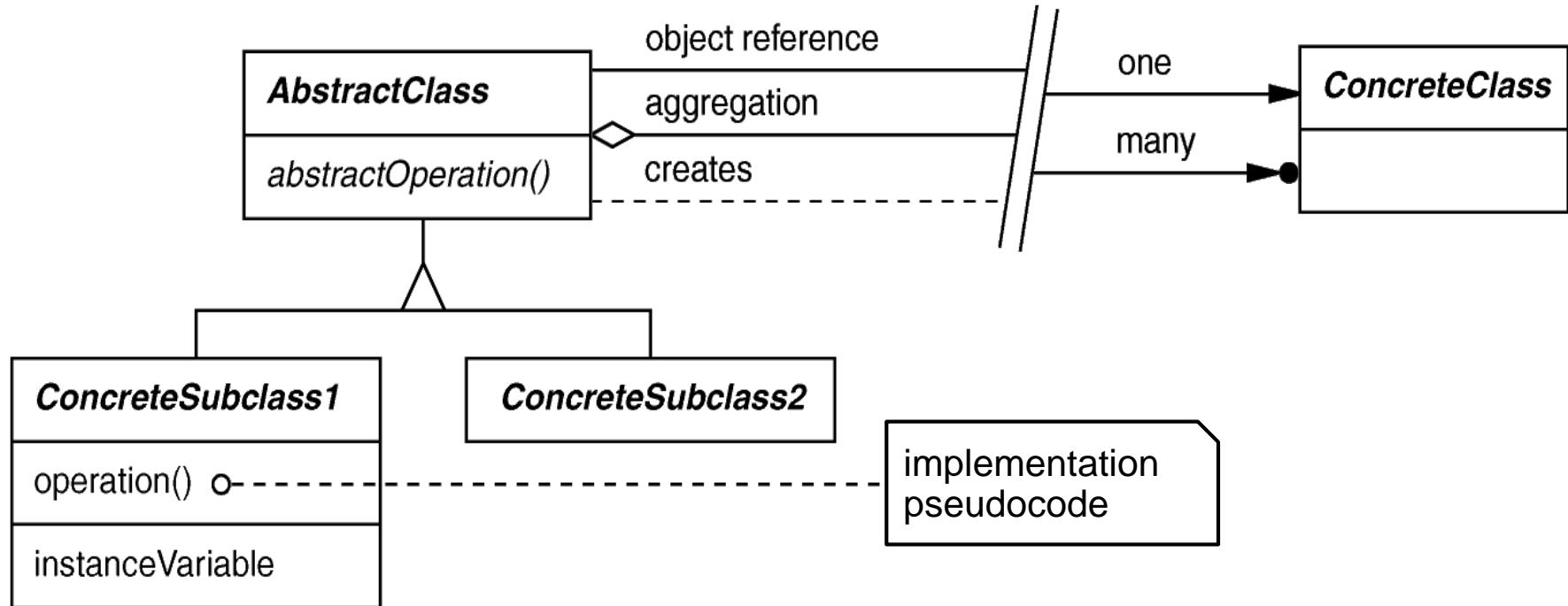
- Put all the pieces together



The Important of Design Experience

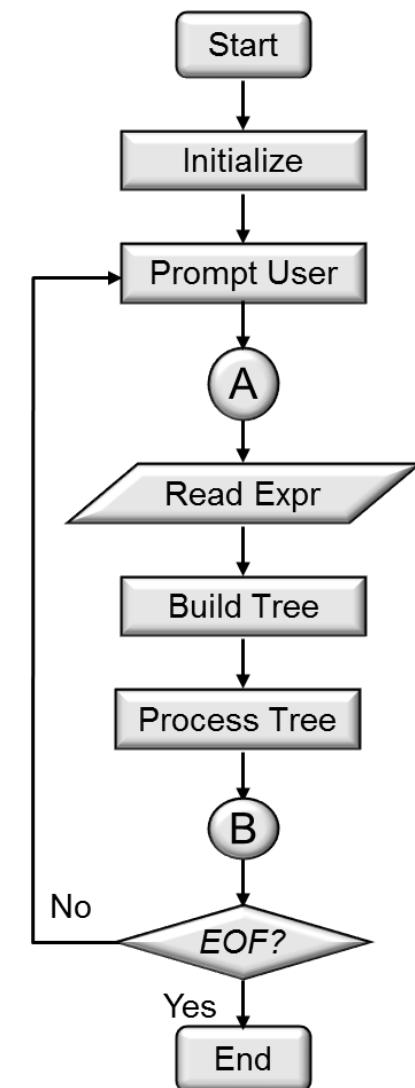
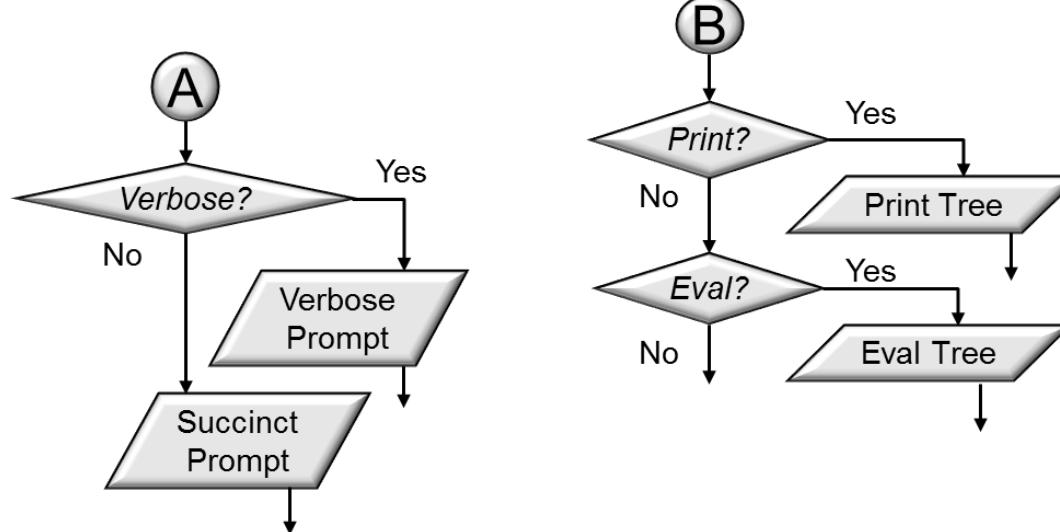
The Importance of Design Experience

- To be a master software developer it's necessary—*but not sufficient*—to have expertise at several things, e.g.
 - Modeling notations** – that visualize software structure & behavior



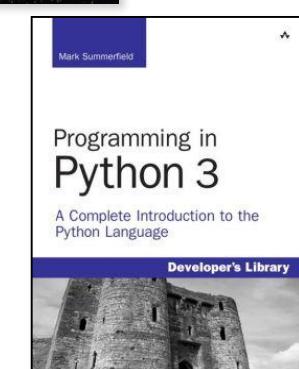
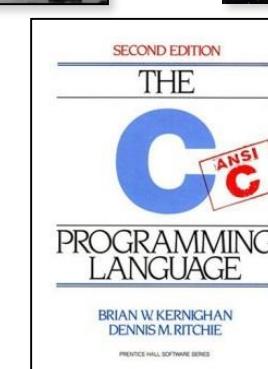
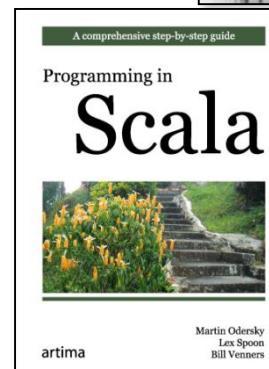
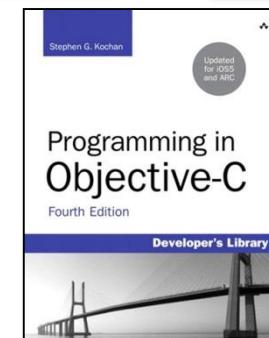
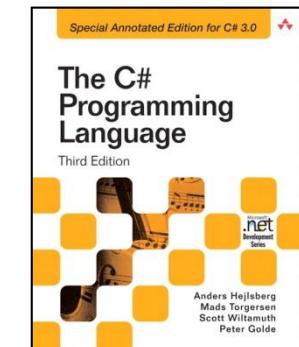
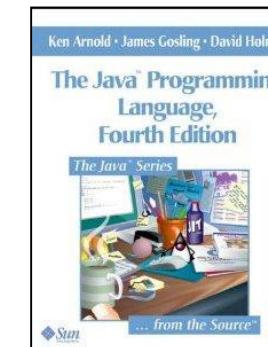
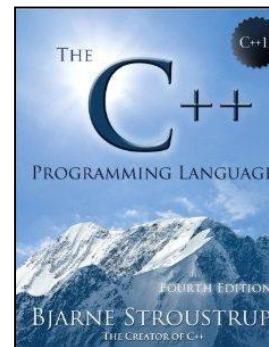
The Importance of Design Experience

- To be a master software developer it's necessary—*but not sufficient*—to have expertise at several things, e.g.
 - Modeling notations**
 - Algorithms** – provide step-by-step procedures for calculations & computations



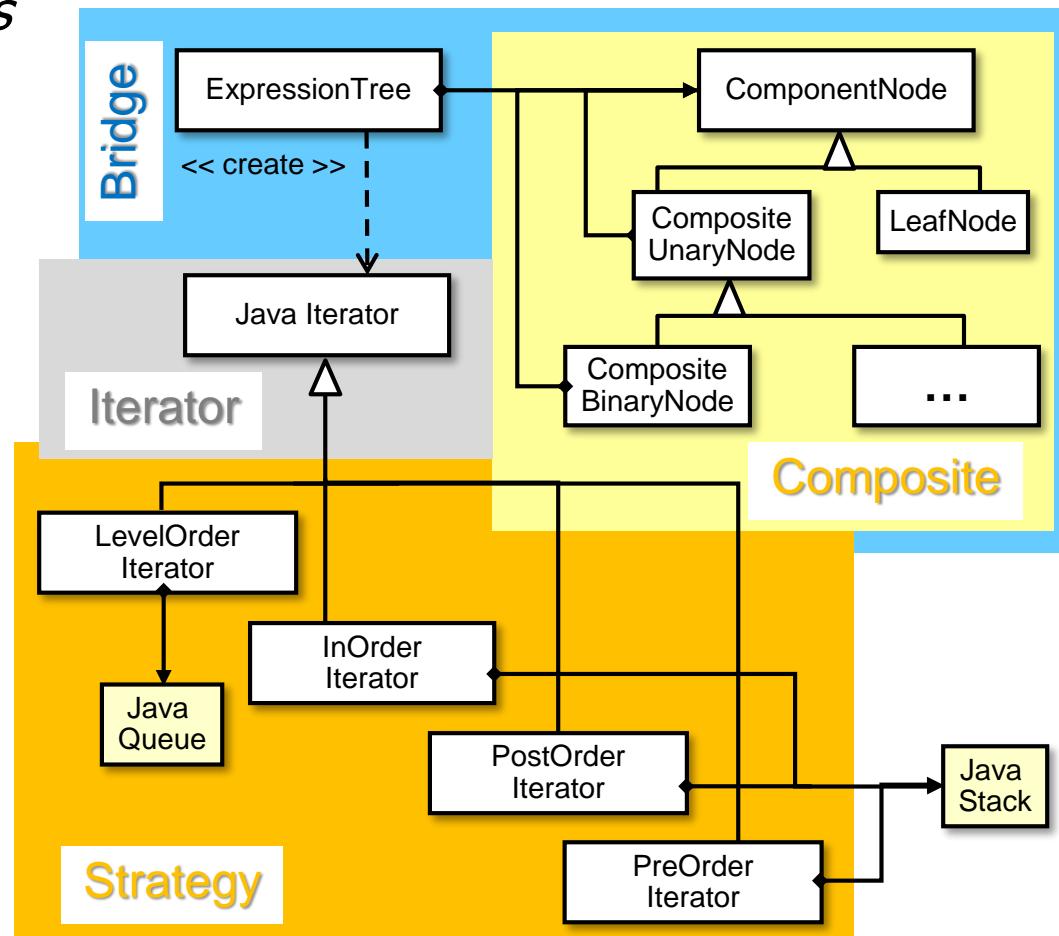
The Importance of Design Experience

- To be a master software developer it's necessary—*but not sufficient*—to have expertise at several things, e.g.
 - Modeling notations
 - Algorithms
 - Programming languages** – define formalisms that control the behavior of computers & expressing algorithms precisely



The Importance of Design Experience

- It's also essential to understand how well-designed software exhibits *recurring structures & behaviors*
 - Knowledge of these recurring structures & behaviors helps enhance key software quality attributes, including...



The Importance of Design Experience

- **Abstraction** – emphasize what's important & de-emphasize the unimportant at a particular level of detail



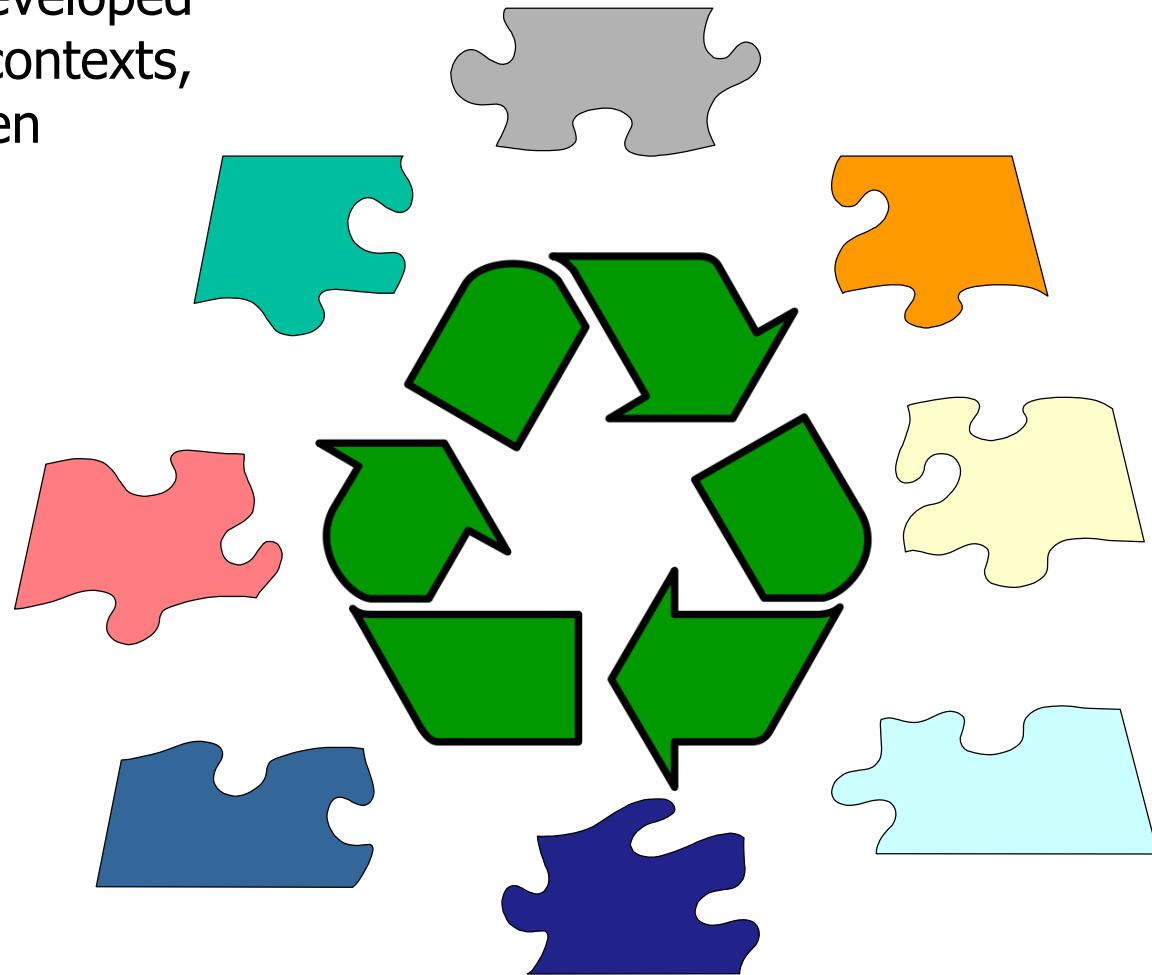
The Importance of Design Experience

- **Flexibility** – apply components in various ways that weren't originally anticipated



The Importance of Design Experience

- **Reuse** – take components developed before & apply them in new contexts, where they may not have been intended initially



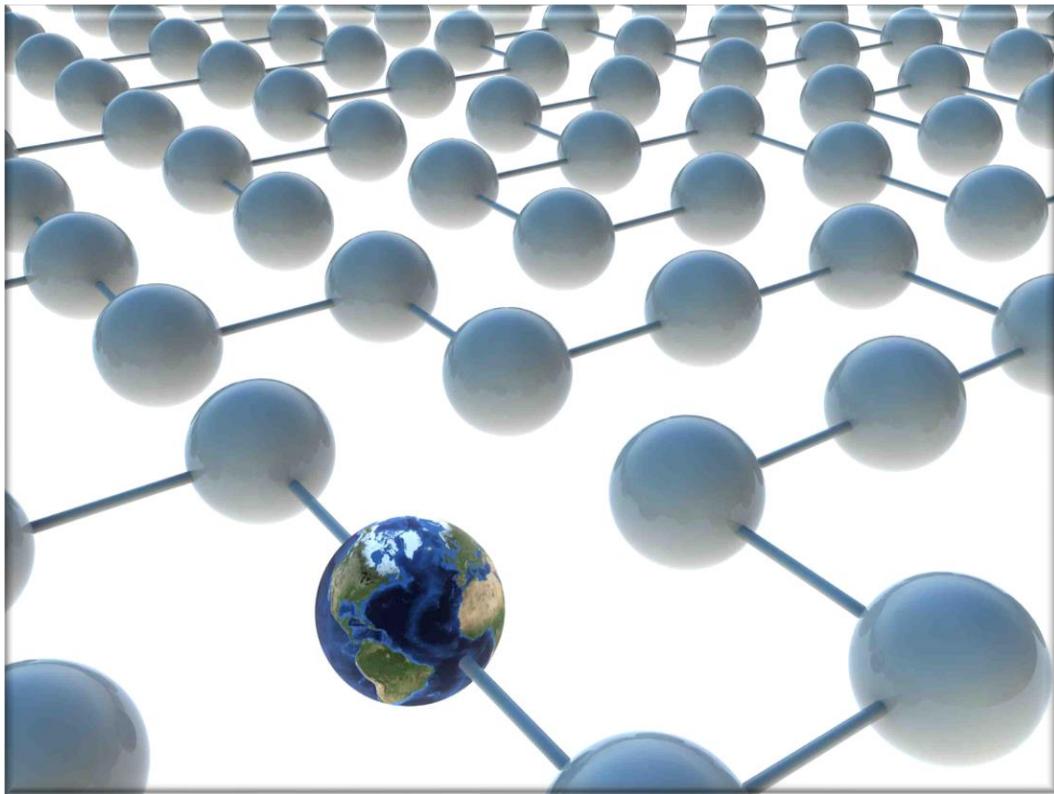
The Importance of Design Experience

- **Quality** – assure that bugs are eliminated, vulnerabilities fixed, & code is made more “bullet proof”



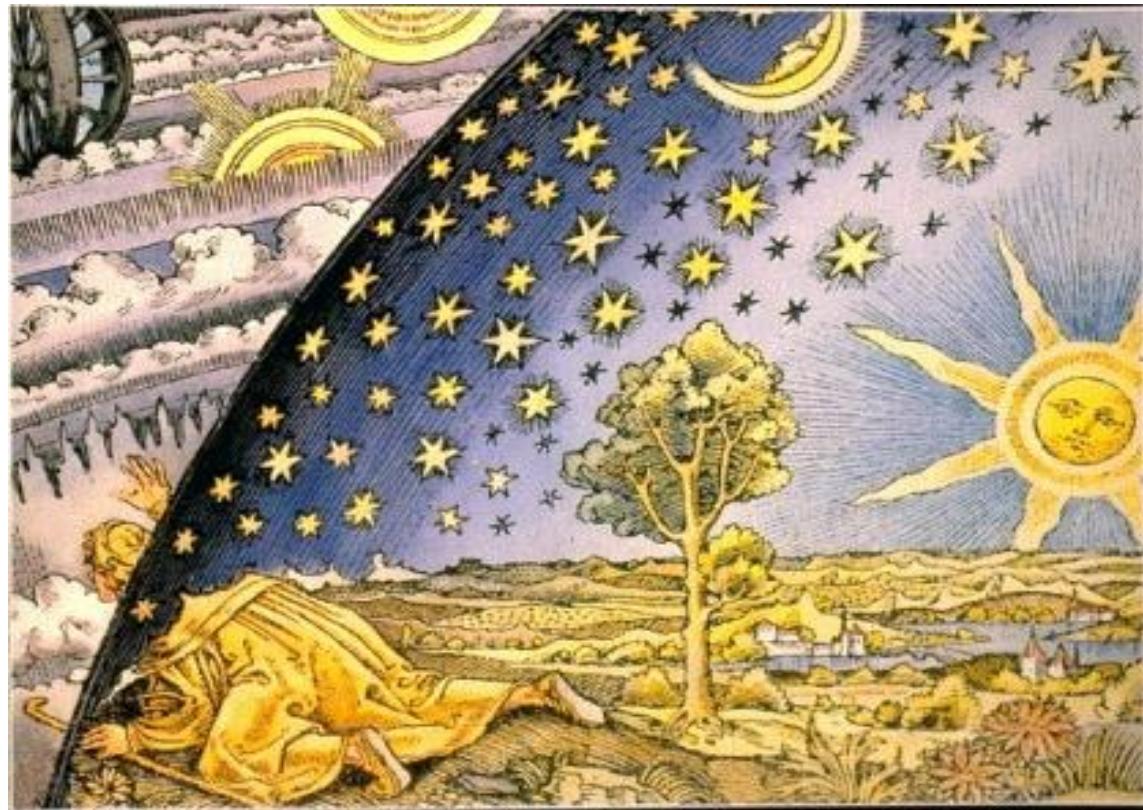
The Importance of Design Experience

- **Modularity** – divide & conquer a big problem space into smaller chunks that have low coupling & high cohesion
 - Allows developers & teams can work independently



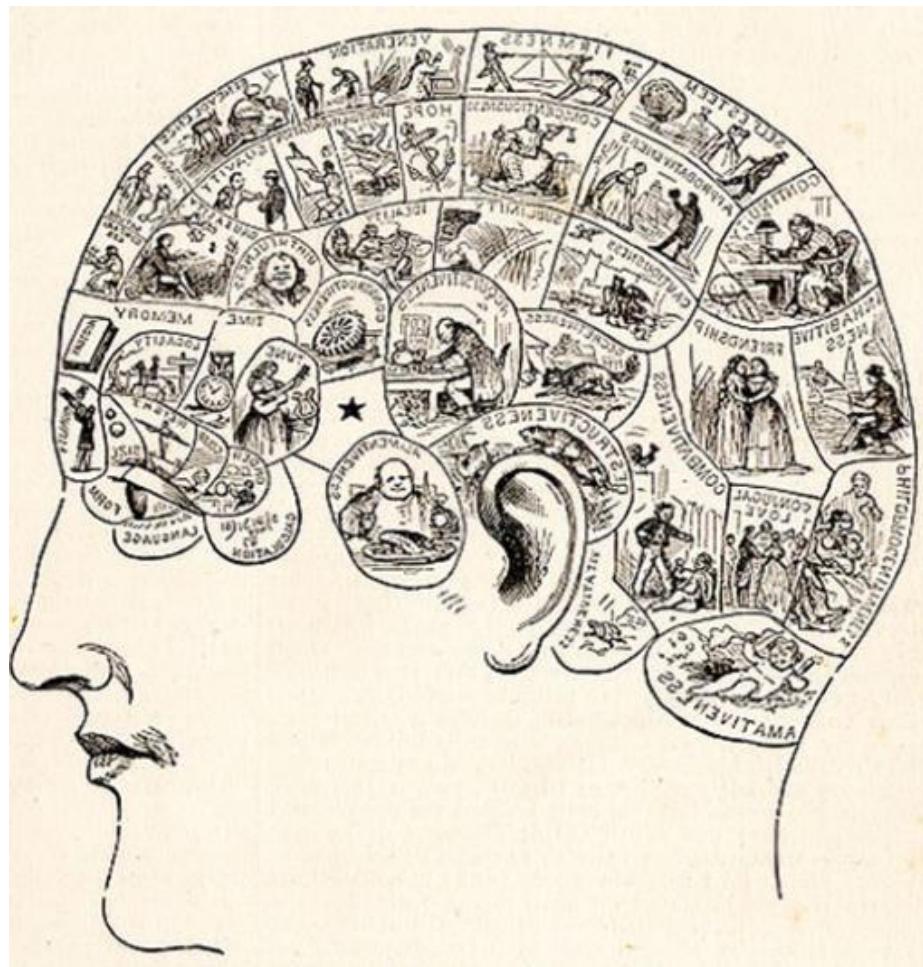
The Importance of Design Experience

- *Valuable design experience* resides in these recurring structures & behaviors



The Importance of Design Experience

- Unfortunately, this design experience has historically been hard to access
 1. *It's in the heads of the experts*



The Importance of Design Experience

- Unfortunately, this design experience has historically been hard to access

1. *It's in the heads of the experts*
2. *It's in the bowels of the source code*

```
public class KeyGeneratorImpl extends Service {  
    private Set<UUID> keys =  
        new HashSet<UUID>();  
    private final KeyGenerator.Stub binder =  
        new KeyGenerator.Stub() {  
            public void setCallback  
                final KeyGeneratorCallback callback) {  
                    UUID id;  
                    synchronized (keys) {  
                        do {  
                            id = UUID.randomUUID();  
                        } while(keys.contains(id));  
                        keys.add(id);  
                    }  
                    final String key = id.toString();  
                    try {  
                        Log.d(getClass().getName(),  
                            "sending key" + key);  
                        callback.sendKey(key);  
                    } catch (RemoteException e) {  
                        e.printStackTrace();  
                    }  
                }  
            };  
  
    public IBinder onBind(Intent intent) {  
        return this.binder;  
    }  
}
```

The Importance of Design Experience

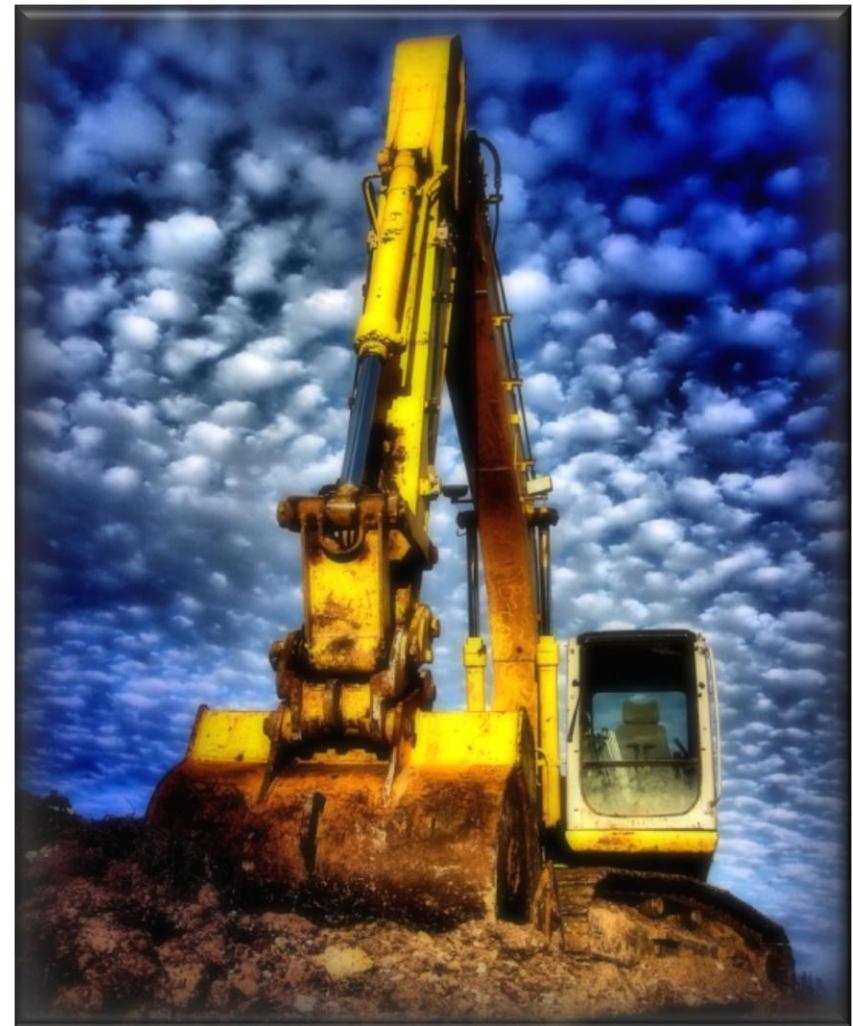
- Unfortunately, this design experience has historically been hard to access
 1. *It's in the heads of the experts*
 2. *It's in the bowels of the source code*



Both locations are
fraught with danger!

The Importance of Design Experience

- What's need is a principled means of *extracting, documenting, conveying, applying, & preserving* expert design experience without undue time, effort, & risk!



Key to Mastery: Knowledge of Design Patterns

Key to Mastery: *Knowledge of Software Patterns*

- Describes a **solution** to a common **problem** arising within a **context**



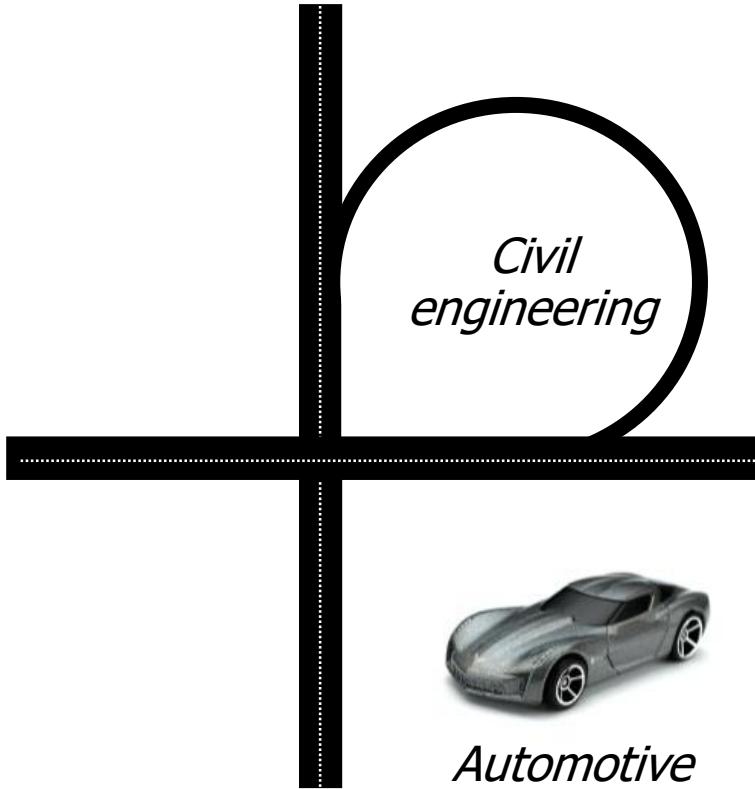
Mobile devices



Aerospace

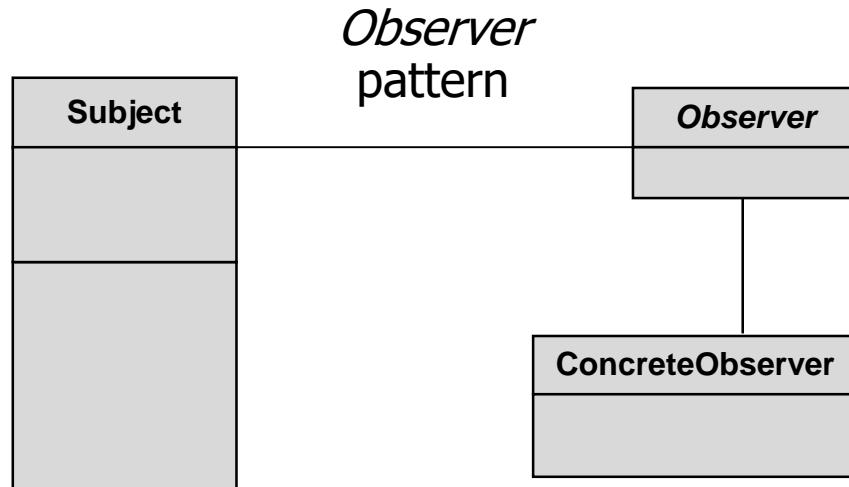


Electronic Trading



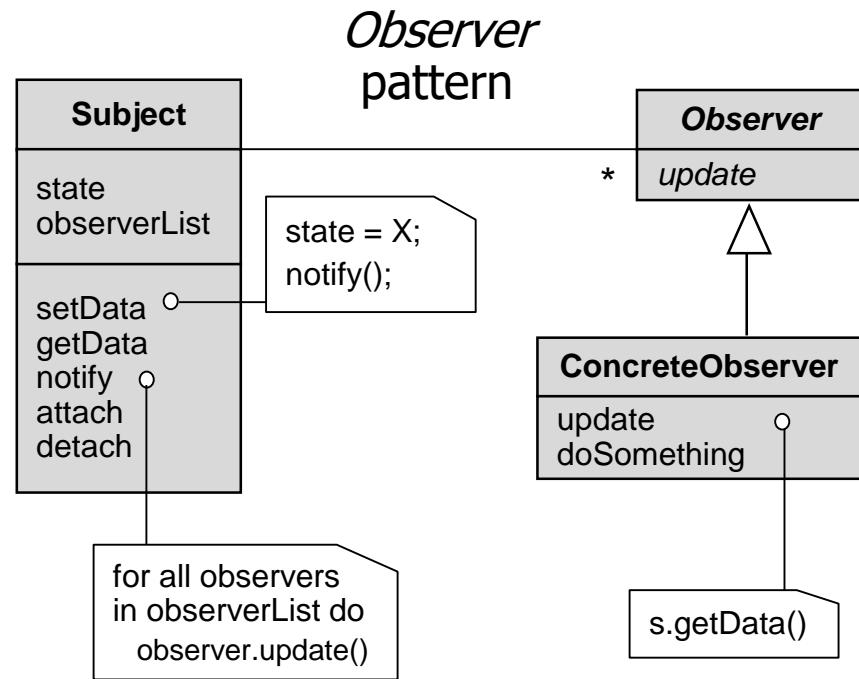
Key to Mastery: *Knowledge of Software Patterns*

- Patterns help improve software quality & developer productivity by
 - **Naming** recurring design structures
 - e.g., the *Observer* pattern “defines a one-to-many dependency between objects so that when one object changes state, all dependents are notified & updated”



Key to Mastery: *Knowledge of Software Patterns*

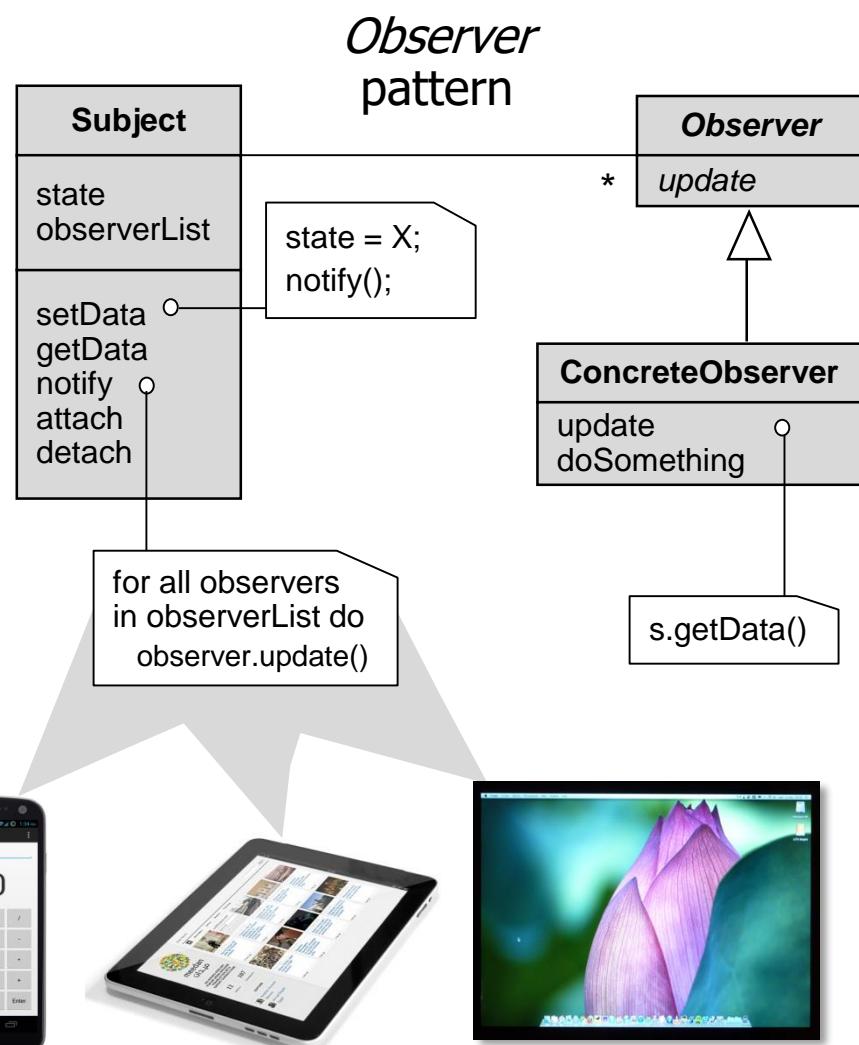
- Patterns help improve software quality & developer productivity by
 - **Specifying** design structure explicitly by identifying key properties of classes & objects, e.g.
 - roles & relationships
 - dependencies
 - interactions
 - conventions



Interpret *class* & *object* loosely: patterns are for more than OO languages!

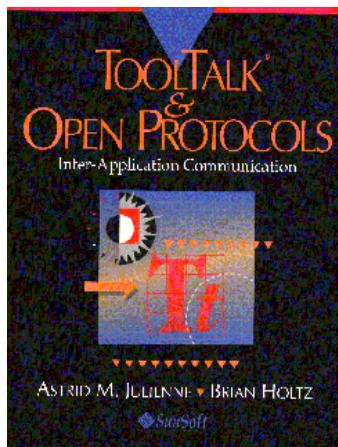
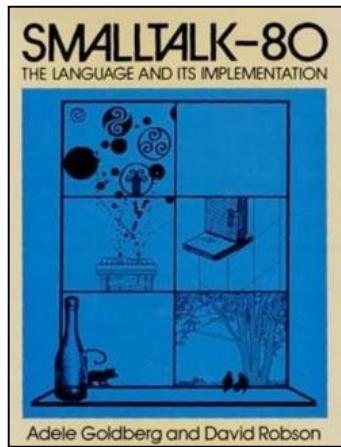
Key to Mastery: *Knowledge of Software Patterns*

- Patterns help improve software quality & developer productivity by
 - **Abstracting** from concrete design elements
 - e.g., problem domain, form factor, vendor, programming language, etc.

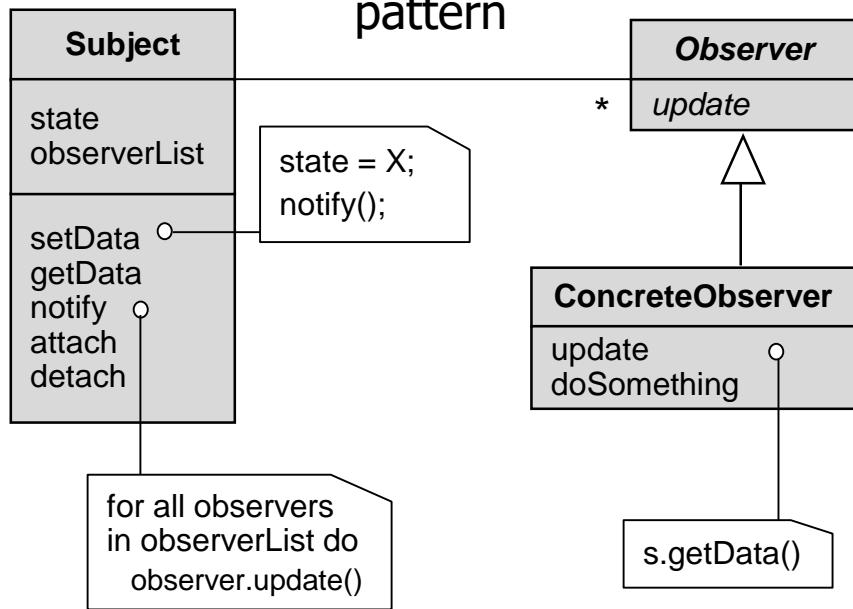


Key to Mastery: *Knowledge of Software Patterns*

- Patterns help improve software quality & developer productivity by
 - **Distilling & codifying knowledge** gleaned from the successful design experience of experts



Observer
pattern

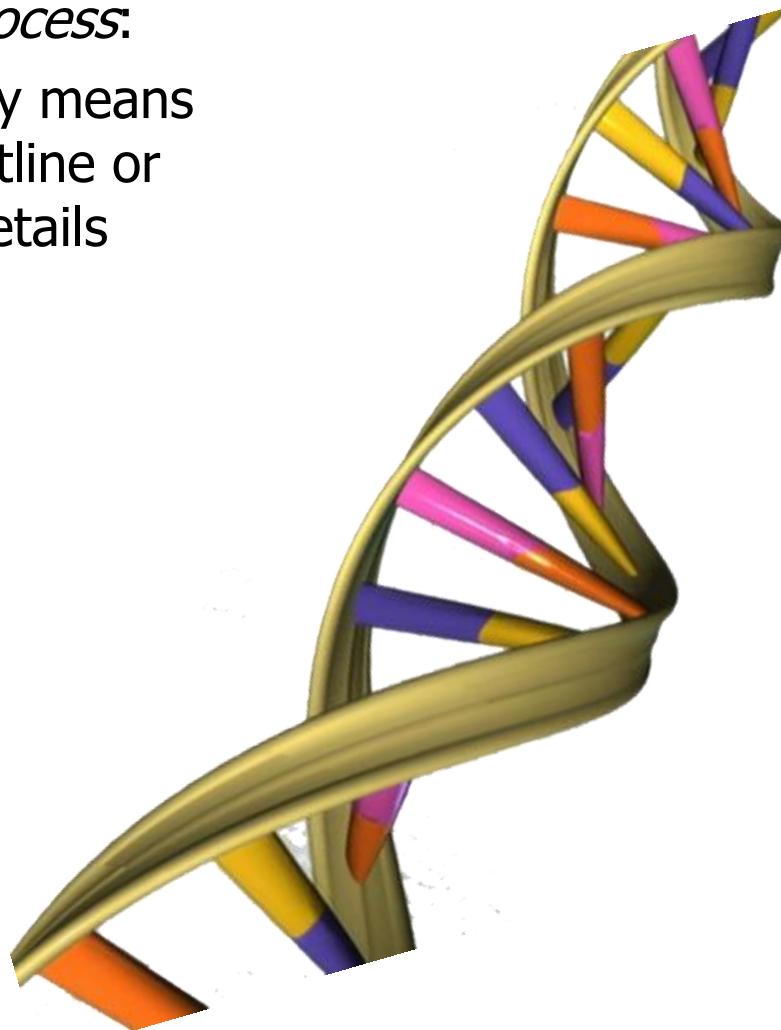


Patterns help avoid “reinventing the wheel” for common software problems

Common Characteristics of Patterns

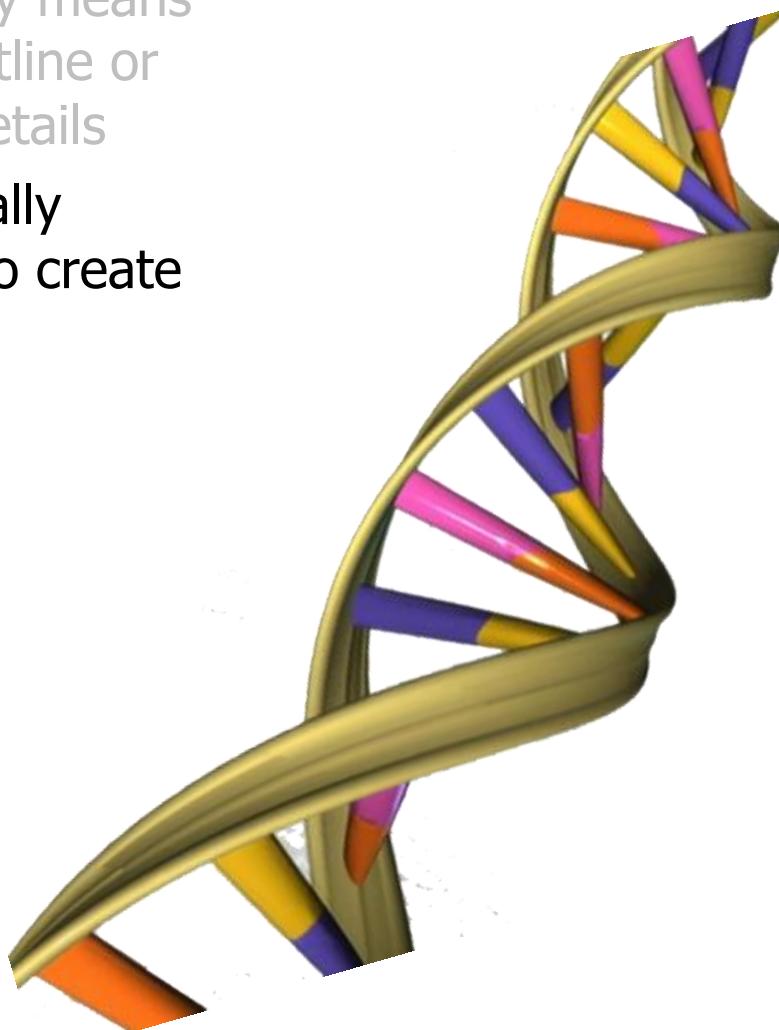
Common Characteristics of Patterns

- They describe both a *thing* & a *process*:
 - The “thing” (the “what”) typically means a particular high-level design outline or description of implementation details



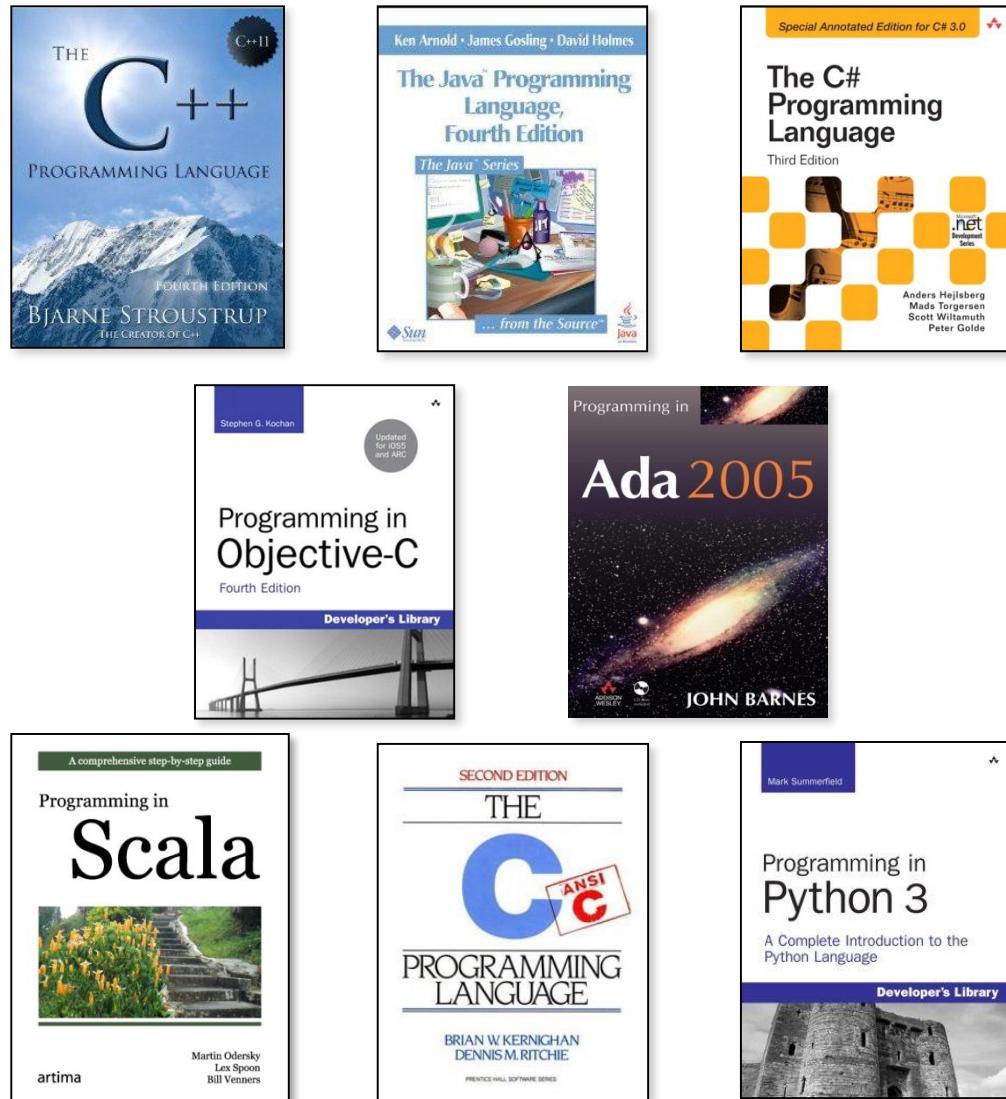
Common Characteristics of Patterns

- They describe both a *thing* & a *process*:
 - The “thing” (the “what”) typically means a particular high-level design outline or description of implementation details
 - The “process” (the “how”) typically describes the steps to perform to create the “thing”



Common Characteristics of Patterns

- They can be independent of programming languages & implementation techniques

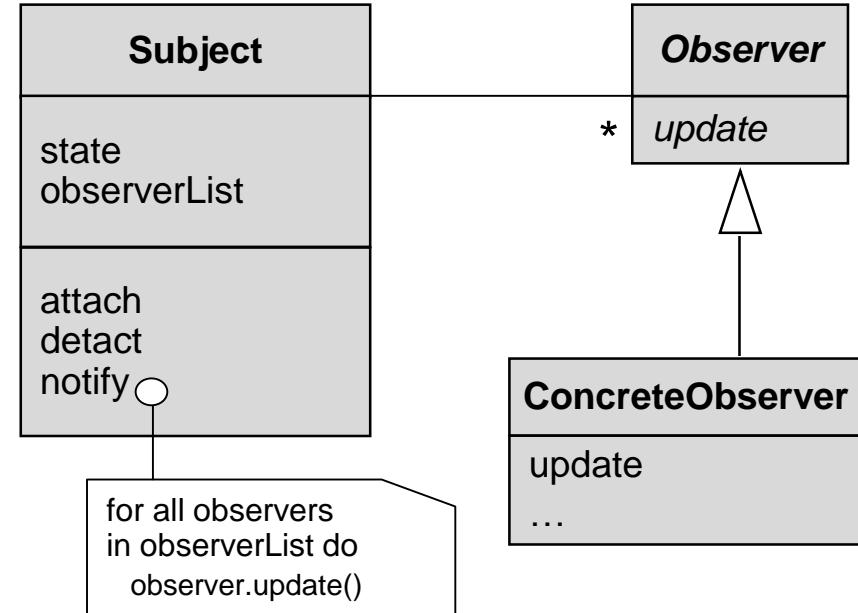


Naturally, (different) patterns apply to different programming languages

Common Characteristics of Patterns

- They define “micro-architectures”
 - In other words, recurring design structure

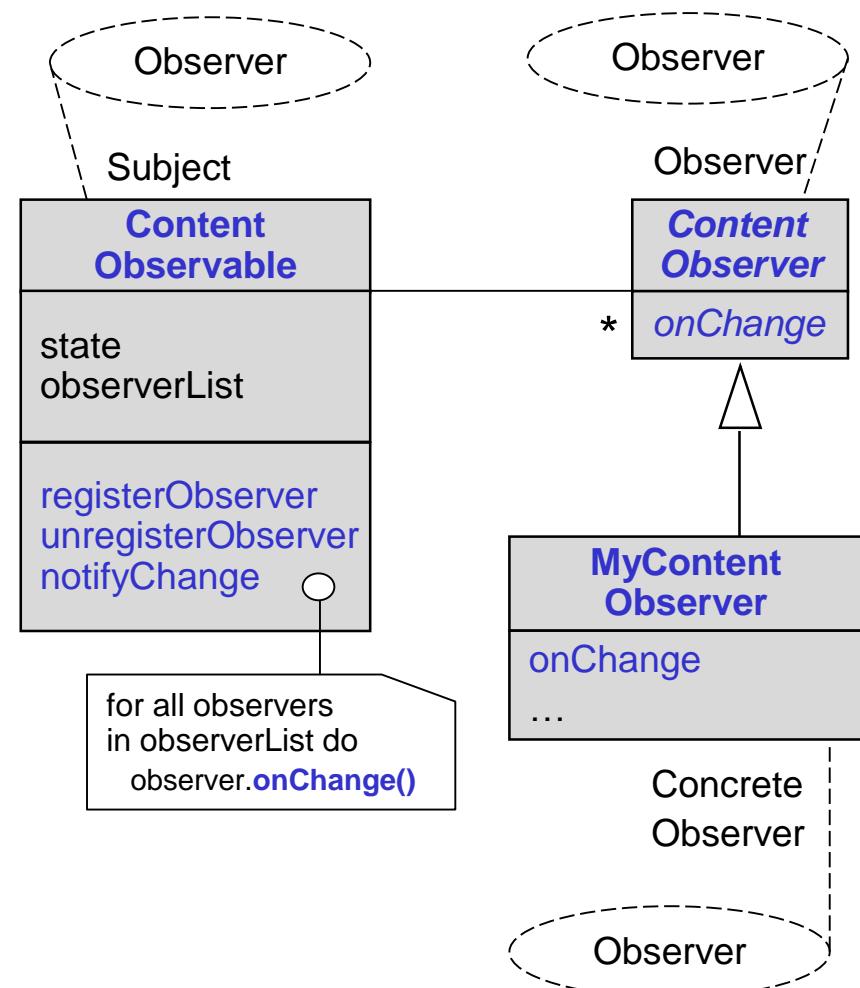
The *Observer* pattern



Common Characteristics of Patterns

- They define “micro-architectures”
 - Certain properties may be modified for particular contexts

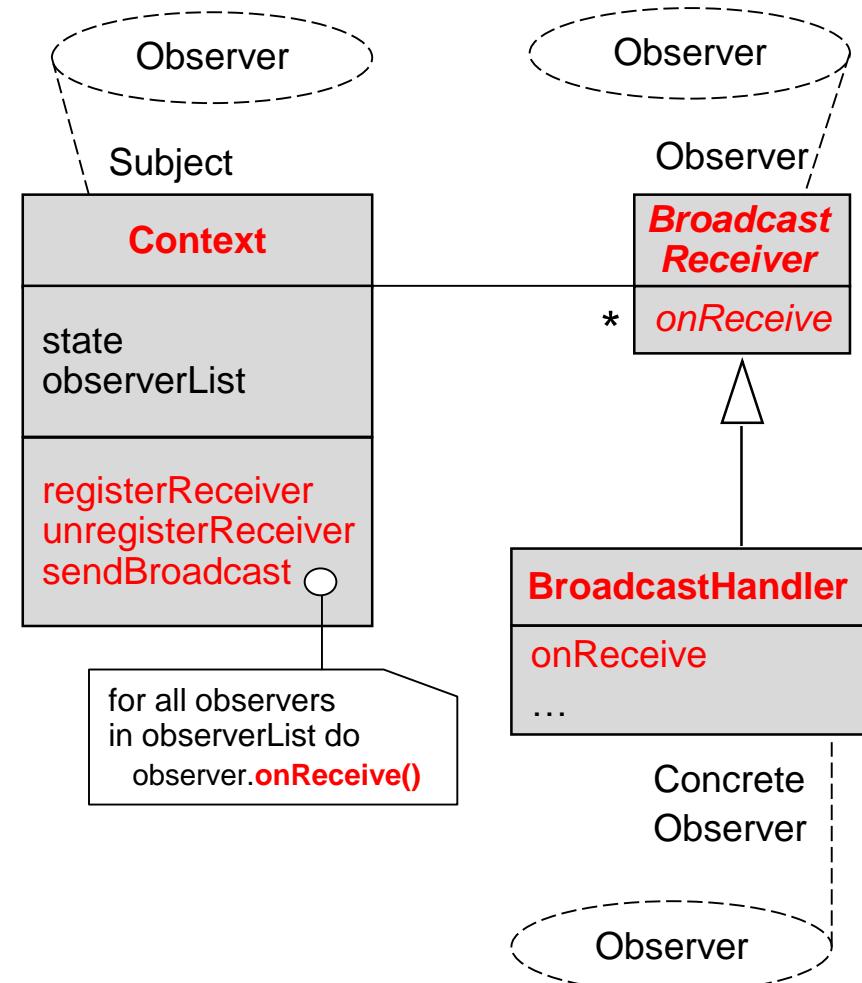
One use of the
Observer pattern
in Android



Common Characteristics of Patterns

- They define “micro-architectures”
 - Certain properties may be modified for particular contexts

A different use of the *Observer* pattern in Android



Common Characteristics of Patterns

- They aren't code or (concrete) designs, so they must be reified & applied in particular languages

The *Observer* pattern in Java

```
public class EventHandler
    extends Observer {
    public void update(Observable o,
                       Object arg)
    { /* */ }
    ...
}

public class EventSource
    extends Observable,
    implements Runnable {
    public void run()
    { /* */ notifyObservers(/* */); }
    ...
}

EventSource eventSource =
    new EventSource();
EventHandler eventHandler =
    new EventHandler();
eventSource.addObserver(eventHandler);
Thread thread =
    new Thread(eventSource);
thread.start();
...
```

Common Characteristics of Patterns

- They aren't code or (concrete) designs, so they must be reified & applied in particular languages

The *Observer* pattern in C++ & ACE

(uses the GoF *Bridge* pattern with reference counting to simplify memory management & ensure exception-safe semantics)

```
class Event_Handler
    : public Observer {
public:
    virtual void update(Observable o,
                         Object arg)
    { /* ... */ }

    ...

class Event_Source
    : public Observable,
      public ACE_Task_Base {
public:
    virtual void svc()
    { /*...*/ notify_observers(/*...*/); }

    ...

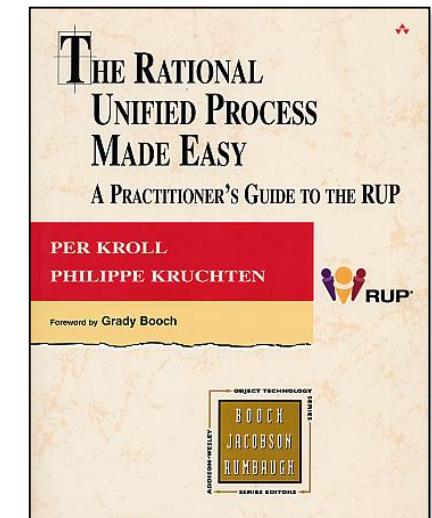
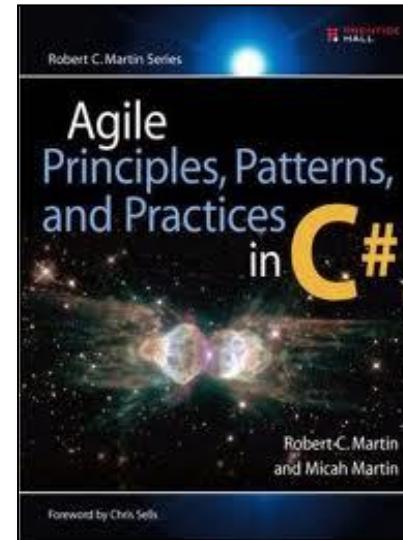
Event_Source event_source;
Event_Handler event_handler;
event_source->add_observer
    (event_handler);

Event_Task task (event_source);
task->activate();

...
```

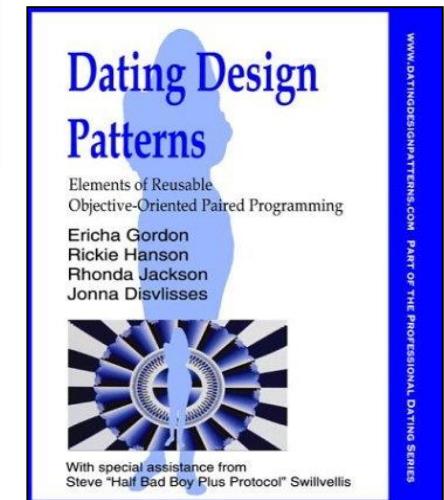
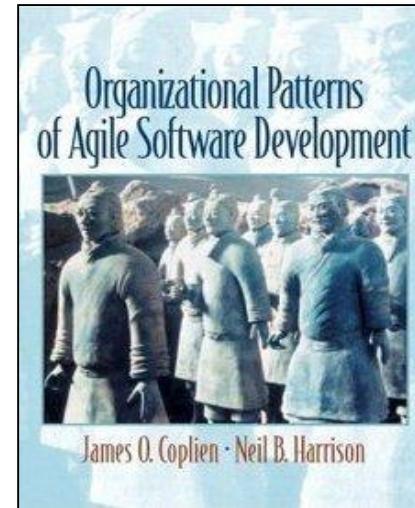
Common Characteristics of Patterns

- They are not methods, but can be used as an adjunct to methods, e.g.:
 - Rational Unified Process
 - Agile
 - Others



Common Characteristics of Patterns

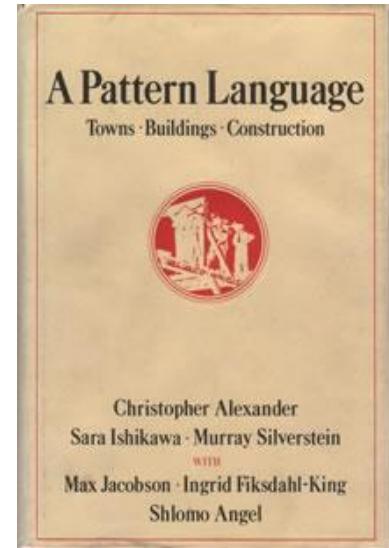
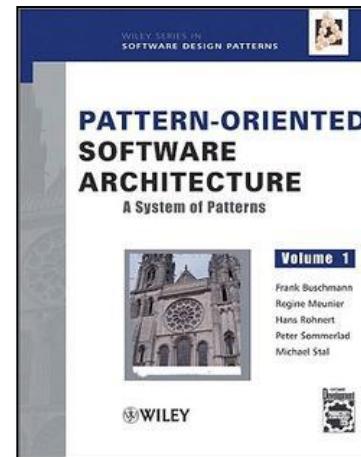
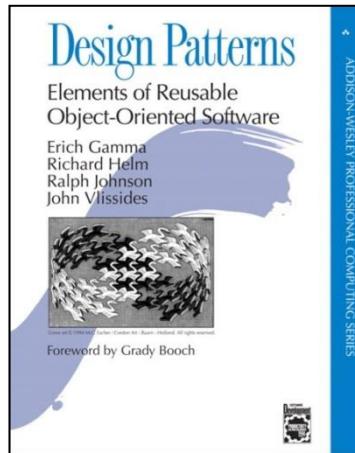
- There are also patterns for organizing effective software development teams & navigating other complex settings



Common Parts of a Pattern Description

Common Parts of a Pattern Description

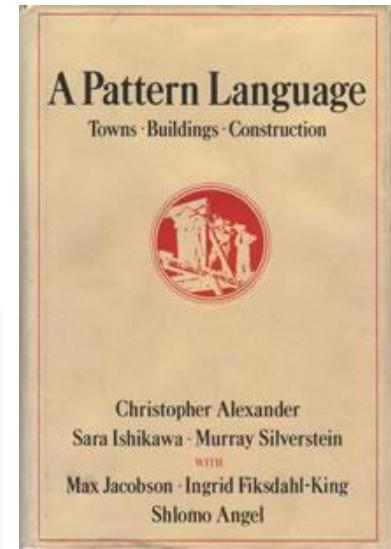
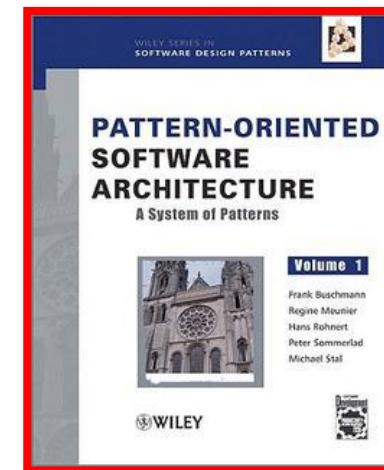
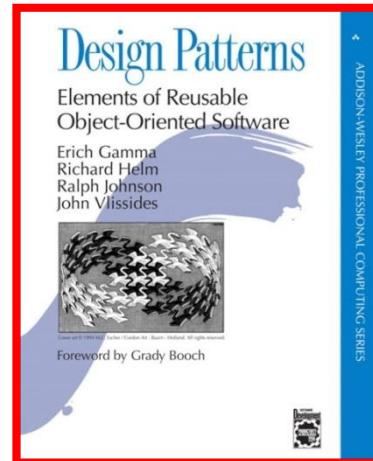
- A range of “pattern forms” have been used to document patterns



See c2.com/cgi/wiki?PatternForms for more info on pattern forms

Common Parts of a Pattern Description

- A range of “pattern forms” have been used to document patterns
 - We’ll focus on the “Gang of Four” (GoF) & POSA pattern forms



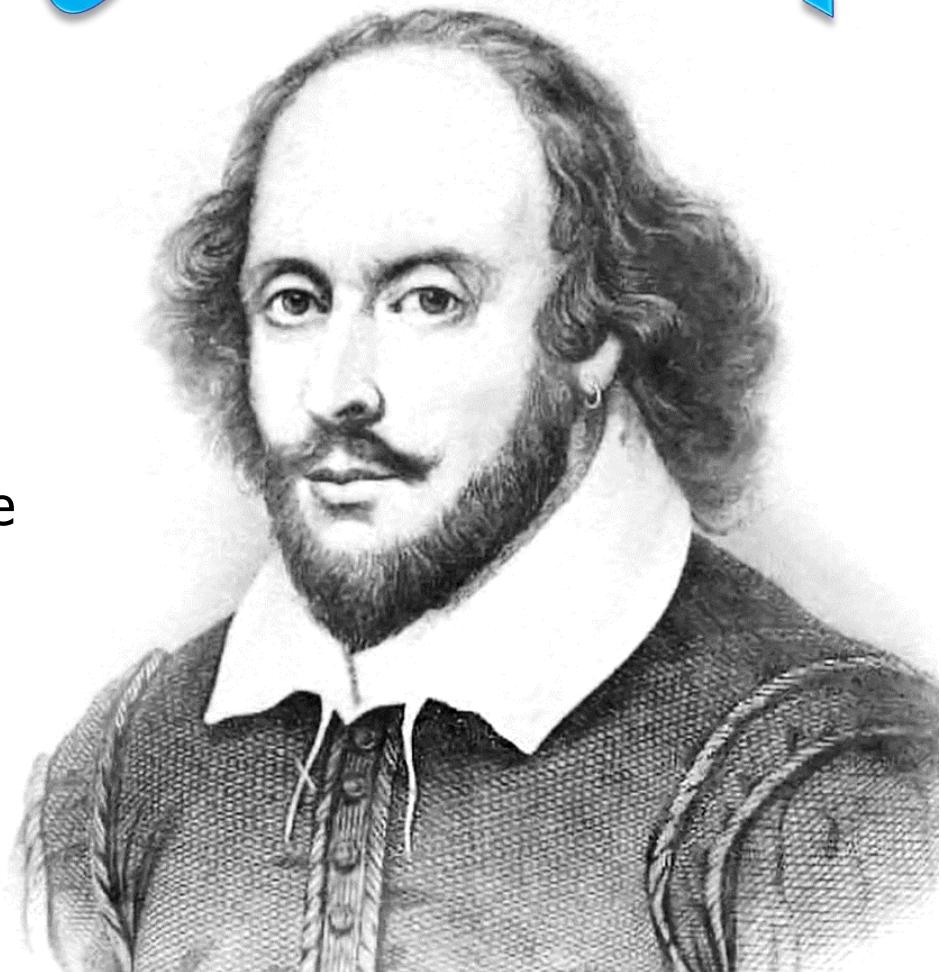
Common Parts of a Pattern Description

- **Name**
 - Should be pithy & memorable

Observer

"What's in a name? That which we call a rose by any other name would smell as sweet."

– William Shakespeare



Common Parts of a Pattern Description

- **Intent**

- Goal behind the pattern & the reason(s) for using it

Observer

Observer: Define a one-to-many dependency between objects so that when one object changes state, all dependents are notified & updated



Common Parts of a Pattern Description

- **Problem** addressed by the pattern
 - Motivate the “forces” & situations in which pattern is applicable

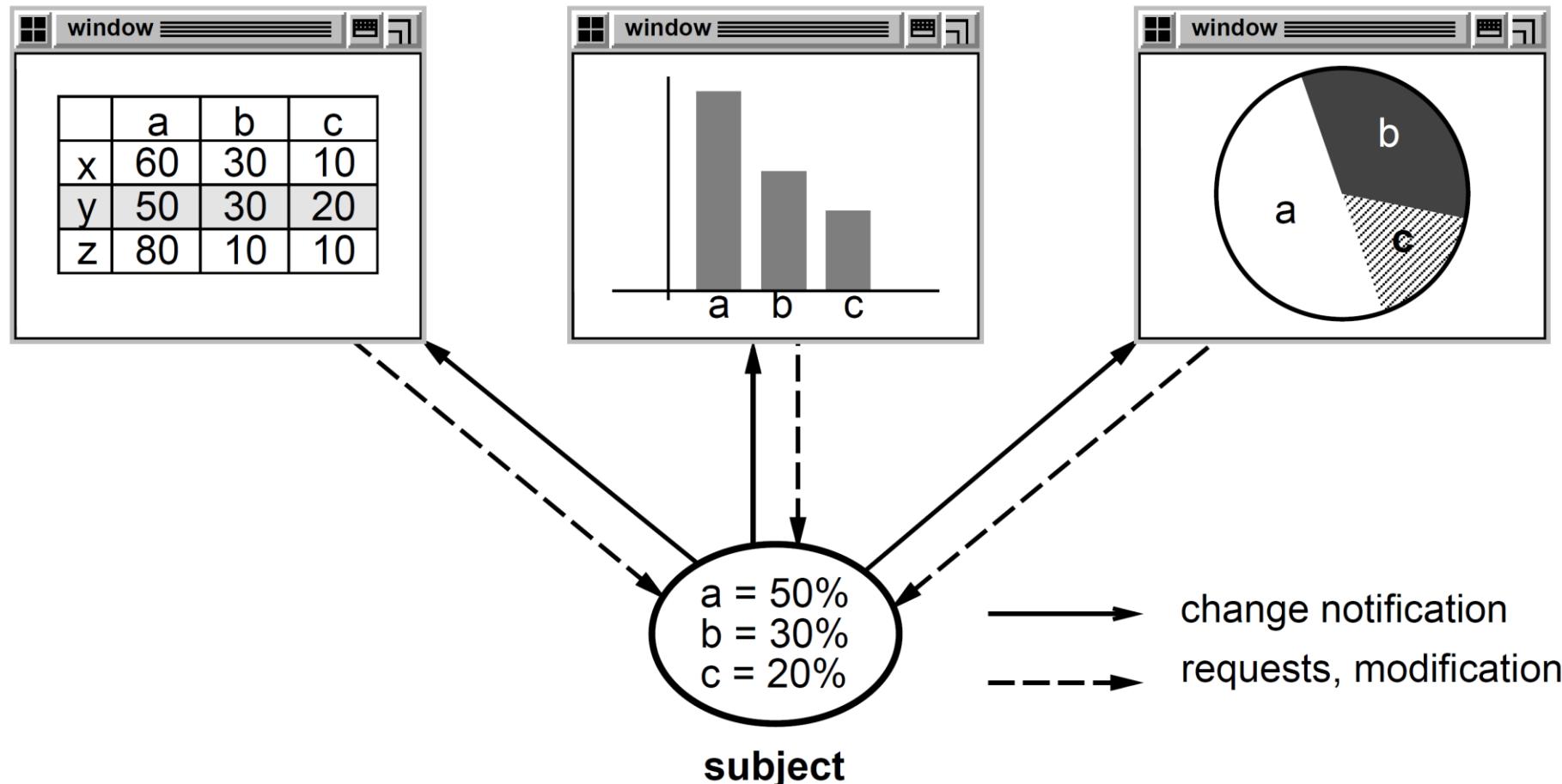
Observer

- An abstraction has two aspects, one dependent on the other
- A change to one object requires changing untold others
- An object should notify unknown other objects

Common Parts of a Pattern Description

- **Problem** addressed by the pattern
 - Motivate the “forces” & situations in which pattern is applicable

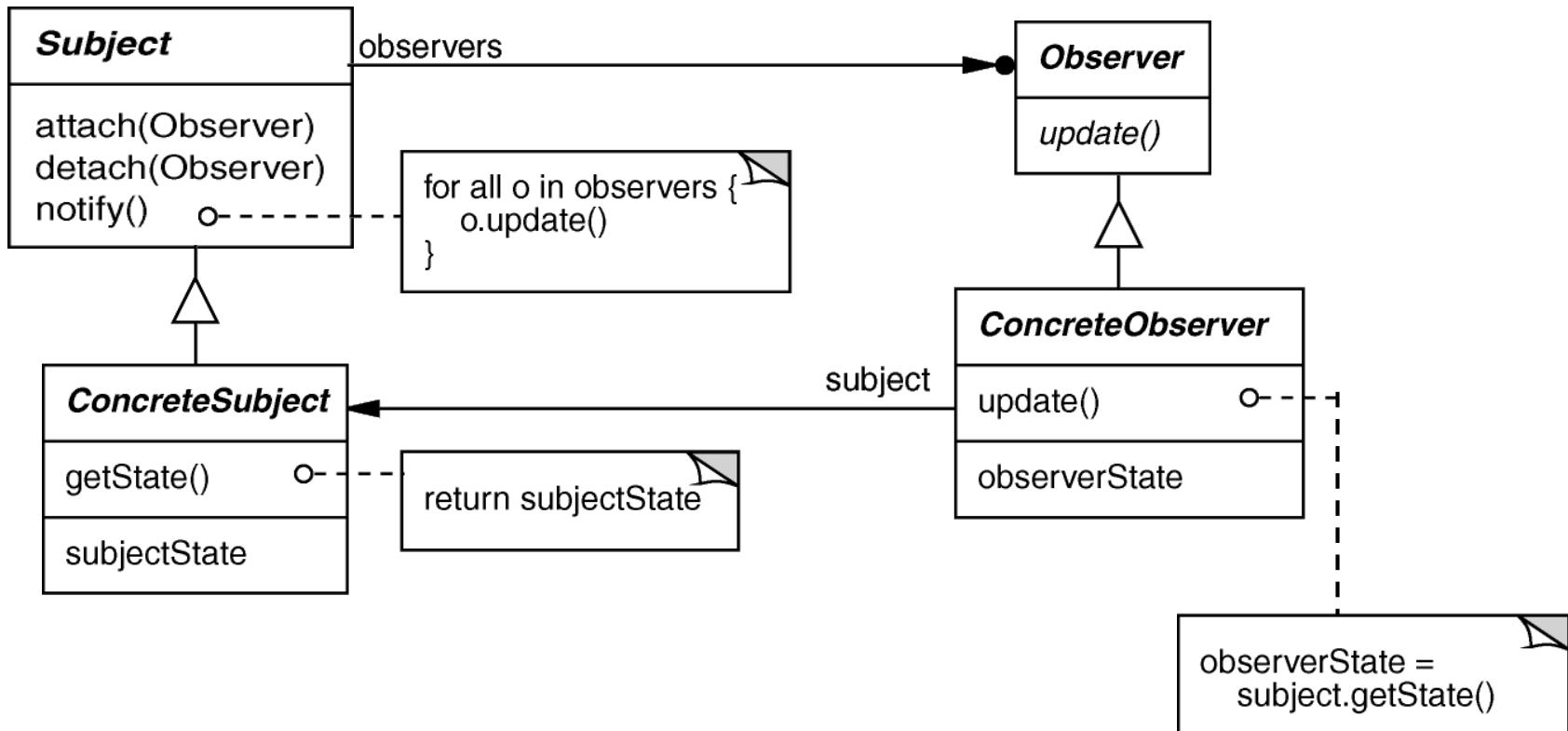
Observer



Common Parts of a Pattern Description

- **Solution** provided by the pattern
 - Visual & textual descriptions of the static structure, participants, & collaboration dynamics of a pattern

Observer

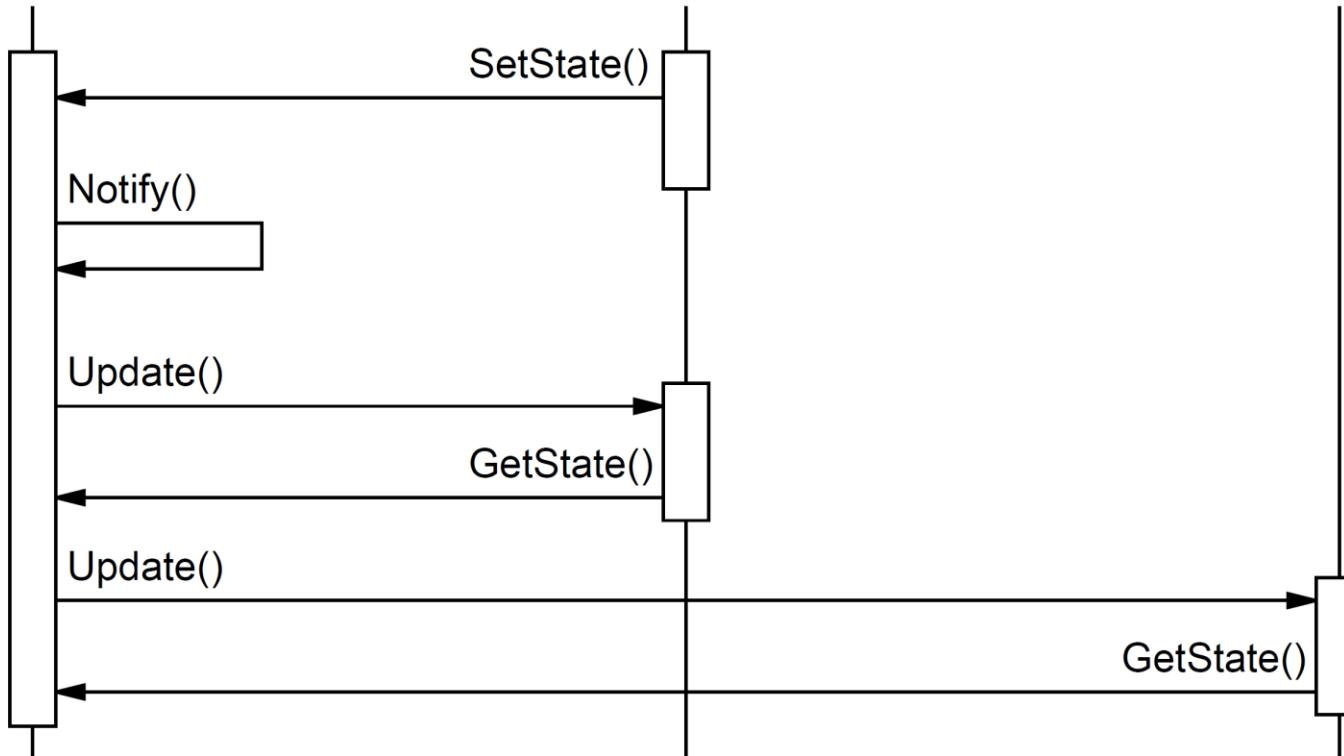


Common Parts of a Pattern Description

- **Solution** provided by the pattern
 - Visual & textual descriptions of the static structure, participants, & collaboration dynamics of a pattern

Observer

aConcreteSubject aConcreteObserver anotherConcreteObserver



Common Parts of a Pattern Description

- **Example implementation guidance**

- e.g., source code snippets in one or more programming languages

```
public class EventHandler
    extends Observer {
public void update
    (Observable o,
     Object arg)
{ /* ... */ }
...
}
```

```
public class EventSource
    extends Observable,
    implements Runnable {
public void run() {
    ...
    for (Observer o : mObservers)
        o.notify(/* ... */);
    ...
}
```



```
EventSource eventSource =
    new EventSource();
EventHandler eventHandler =
    new EventHandler();
eventSource.addObserver
    (eventHandler);
Thread thread =
    new Thread(eventSource);
thread.start();
...
}
```

Common Parts of a Pattern Description

- **Consequences**

- Benefits & liabilities of applying the pattern



Observer

- + modularity: subject & observers may vary independently
- + extensibility: can define & add any number of observers
- + customizability: different observers offer different views of subject
- unexpected updates: observers don't know about each other
- update overhead: might need hints or filtering

Common Parts of a Pattern Description

- **Known uses**

- Examples of real pattern uses



Observer

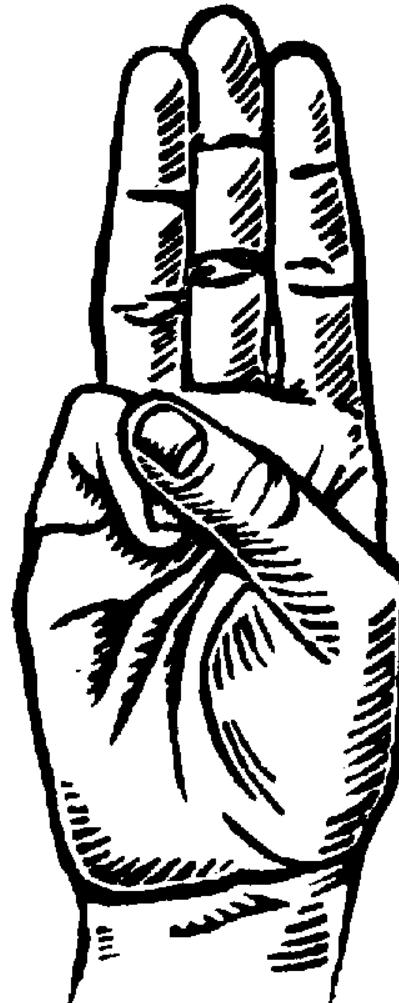
- Smalltalk Model-View-Controller (MVC)
- InterViews (Subjects & Views, Observer/Observable)
- Java Observer/Observable
- Pub/sub middleware (e.g., CORBA Notification Service, Data Distribution Service, Java Message Service)
- Mailing lists

Common Parts of a Pattern Description

- **Known uses**

- Examples of real pattern uses
- Should follow the “rule of three”

Observer



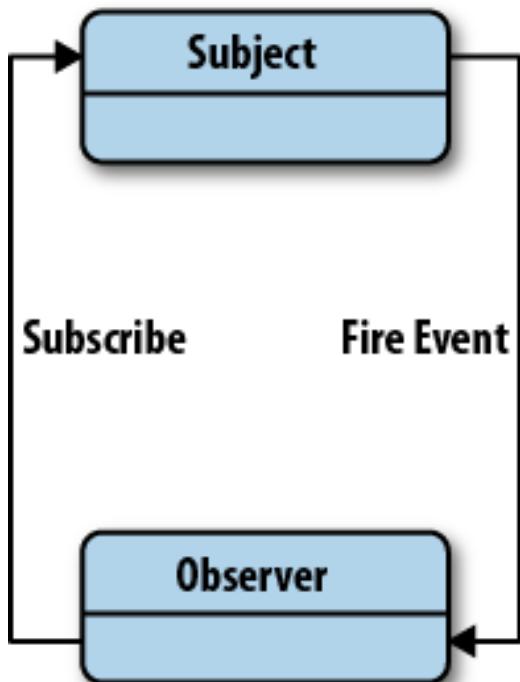
Common Parts of a Pattern Description

- **Related patterns**

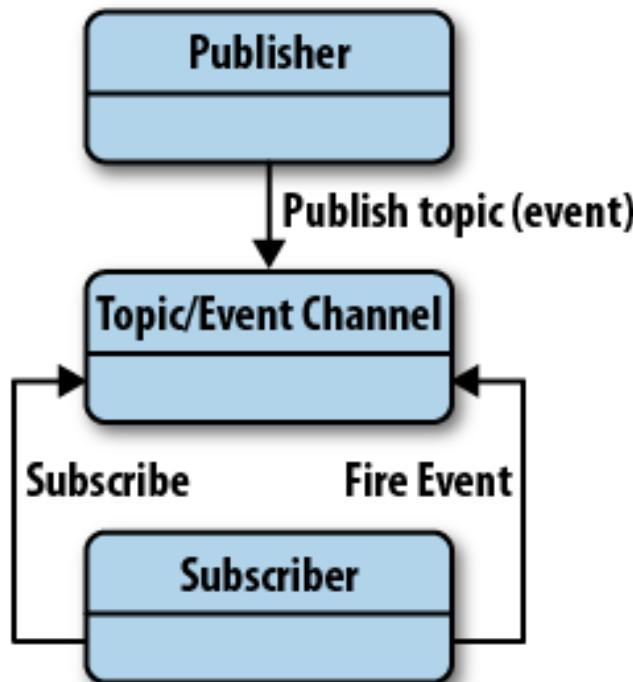
- Summarize relationships & tradeoffs between alternative patterns for similar problems

Observer

Observer Pattern



Publish/Subscribe



Common Parts of a Pattern Description

- **Related patterns**

- Summarize relationships & tradeoffs between alternative patterns for similar problems



Stand-alone “pattern islands” are unusual in practice

Brief History of the Gang-of-Four Book

Brief History of the Gang-of-Four Pattern Book

- 1991 – Erich Gamma completes his PhD dissertation on patterns for GUIs



Brief History of the Gang-of-Four Pattern Book

- 1992-1993 – Gang-of-Four attend OOPSLA workshops on “Towards an Architecture Handbook”

The image shows the cover of the "OOPSLA '92" workshop report. The cover features a black and white photograph of a mountain range at the top. Below the photo, the text "OOPSLA '92" is written in a large, bold, sans-serif font. Underneath that, it says "Vancouver, British Columbia, Canada" and "5 - 10 October 1992". To the right of the main title, there is a smaller section titled "Addendum to the Proceedings". The central part of the cover has a thin horizontal line separating the header from the main content area. Inside this area, the title "Workshop Report—Towards an Architecture Handbook" is centered in a bold, sans-serif font. Below this title, there is a section titled "Report by:" followed by the author's name, "Bruce Anderson, University of Essex". The main body of the report is divided into several sections: "Introduction", "Some Context", "Architecture", and "Handbooks". Each section contains a block of text. To the right of the "Introduction" section, there is a note stating: "We do not define architecture. Like Health or Quality, we can reach a productive shared understanding for the purpose at hand without making a definition. Different groups will have different understandings, and these will be reflected in different practices: processes, checklists, libraries, diagrams. And different handbooks." Below this, another note states: "Note also that a system may contain several architectures, and that it may have a different architecture when viewed from a different perspective. Many of our powerful yet ambiguous words may be brought into play: 'we are using an object-oriented paradigm to build a generic-application framework via a delegation approach.'". At the bottom of the page, there is a footer with the date "5-10 October 1992", the page number "- 109 -", and the text "Addendum to the Proceedings".

Report by:
Bruce Anderson,
University of Essex

Introduction
This report describes the workshop, but also describes the context in which it took place. We did not use the workshop (nor its precursor at OOPSLA '91) to present our work (which we did by reading the position papers), but to work together. Such an environment does not automatically produce outputs useful to those who did not attend, but is a source of emerging ideas, new approaches, and new relationships. These ideas, approaches and relationships generate the context described here.

Some Context
Architecture
We take architecture to mean the structuring paradigms, styles and patterns that make up our software systems. They are overarching or pervasive structures.
They are important in many ways: they allow us to talk usefully about systems without talking about their detail; a knowledge of them gives us design choices; attention to this level can make systems have the properties we want, especially reusability and extensibility.
Of course every system has an architecture, but if it has not been consciously created then the system may well be more muddled, both conceptually and in its code, than is necessary.

We do not define architecture. Like Health or Quality, we can reach a productive shared understanding for the purpose at hand without making a definition. Different groups will have different understandings, and these will be reflected in different practices: processes, checklists, libraries, diagrams. And different handbooks.

Note also that a system may contain several architectures, and that it may have a different architecture when viewed from a different perspective. Many of our powerful yet ambiguous words may be brought into play: "we are using an object-oriented paradigm to build a generic-application framework via a delegation approach."

Handbooks
Once some level of consensus is reached in a community, it is possible, and worthwhile, to produce a handbook. So a company (for internal use) or a publisher (for sale) can find someone (an editor) to put together such a book. The source materials for such a book would be found in the community; there would be technical issues of choosing versions and values, but these would come mainly from the already-existing consensus of practitioners.

There is currently no handbook of software architecture available, because at the moment there is no universal, or even large, community of software architects, so any architecture handbook would have to be based instead on the theories of its author.

5-10 October 1992 - 109 - Addendum to the Proceedings

c2.com/cgi/wiki?ArchitectureHandbookWorkshop has more info

Brief History of the Gang-of-Four Pattern Book

- 1993 – Gang-of-Four publish their first paper on patterns at ECOOP

Design Patterns: Abstraction and Reuse of Object-Oriented Design

Erich Gamma^{1*}, Richard Helm², Ralph Johnson³, John Vlissides²

¹ Taligent, Inc.
10725 N. De Anza Blvd., Cupertino, CA 95014-2000 USA

² I.B.M. Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598 USA

³ Department of Computer Science
University of Illinois at Urbana-Champaign
1034 W. Springfield Ave., Urbana, IL 61801 USA

Abstract. We propose design patterns as a new mechanism for expressing object-oriented design experience. Design patterns identify, name, and abstract common themes in object-oriented design. They capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities. Design patterns play many roles in the object-oriented development process: they provide a common vocabulary for design, they reduce system complexity by naming and defining abstractions, they constitute a base of experience for building reusable software, and they act as building blocks from which more complex designs can be built. Design patterns can be considered reusable micro-architectures that contribute to an overall system architecture. We describe how to express and organize design patterns and introduce a catalog of design patterns. We also describe our experience in applying design patterns to the design of object-oriented systems.

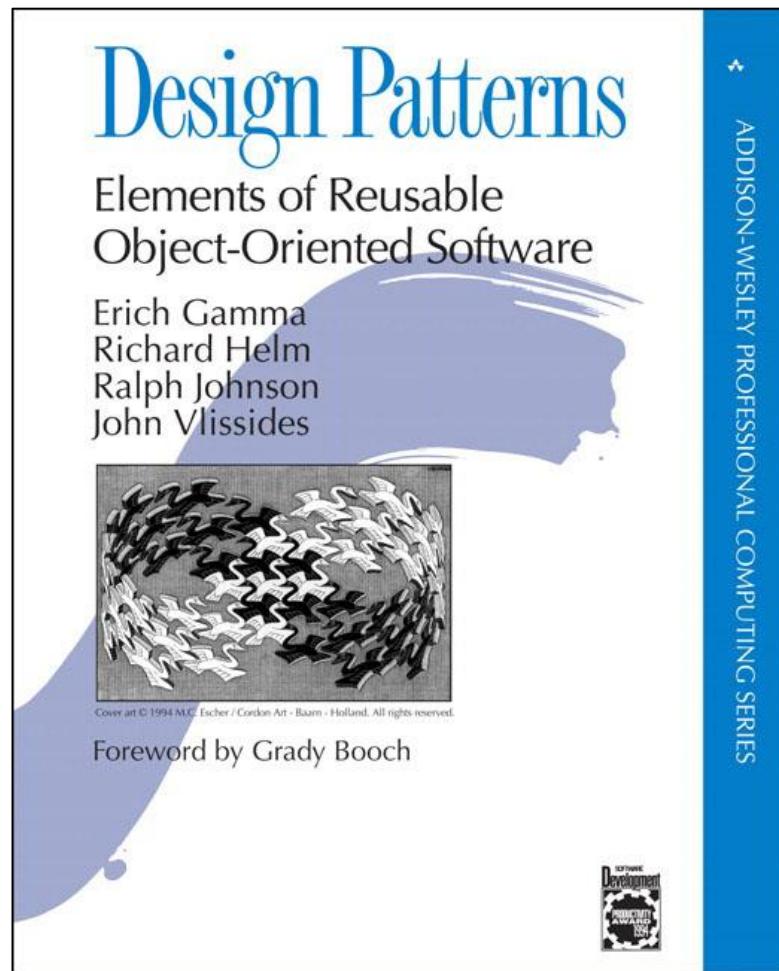
1 Introduction

Design methods are supposed to promote good design, to teach new designers how to design well, and to standardize the way designs are developed. Typically, a design method comprises a set of syntactic notations (usually graphical) and a set of rules that govern how and when to use each notation. It will also describe problems that occur in a design, how to fix them, and how to evaluate a design. Studies of expert programmers for conventional languages, however, have shown that knowledge is not organized simply around syntax, but in larger conceptual structures such as algorithms, data structures and idioms [1, 7, 9, 27], and plans that indicate steps necessary to fulfill a particular goal [26]. It is likely that designers do not think about the notation they are using for recording the design. Rather, they look for patterns to match against plans, algorithms, data structures, and idioms they have learned in the past. Good designers, it appears, rely

* Work performed while at UBILAB, Union Bank of Switzerland, Zurich, Switzerland.

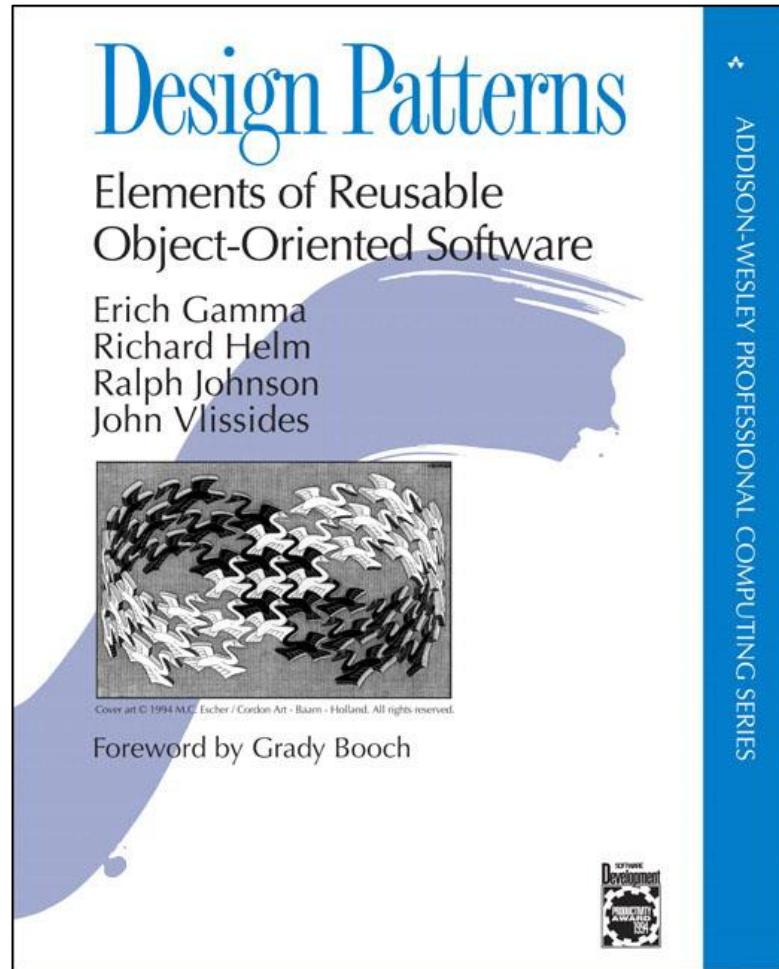
Brief History of the Gang-of-Four Pattern Book

- 1994 – *Design Patterns: Elements of Reusable Object-Oriented Software* (“GoF book”) published



Brief History of the Gang-of-Four Pattern Book

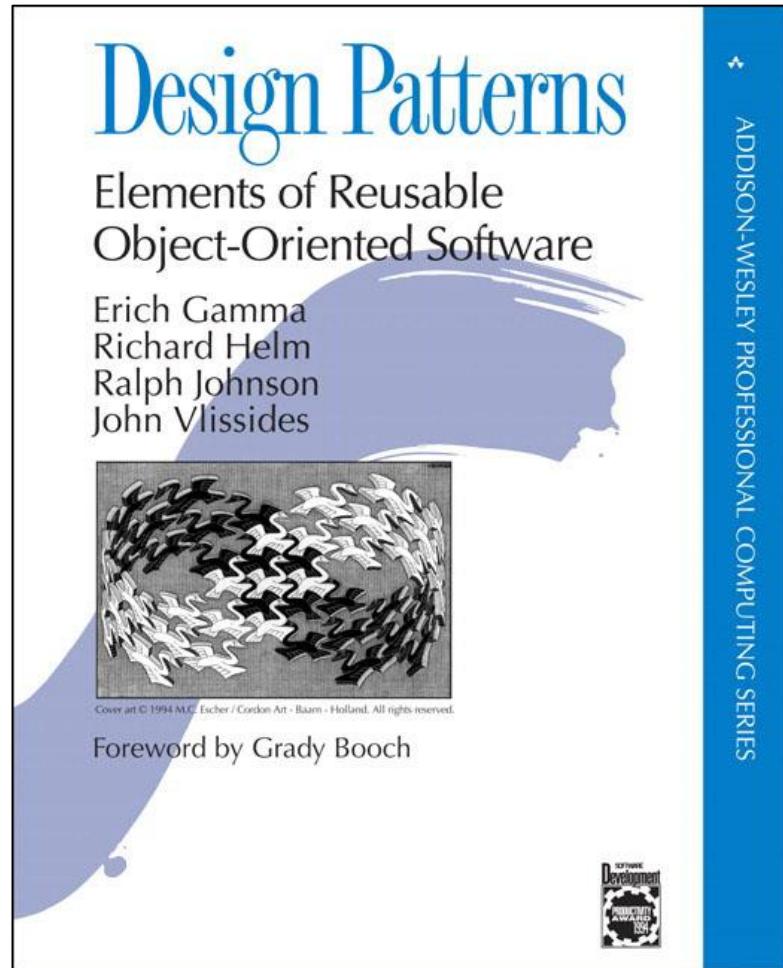
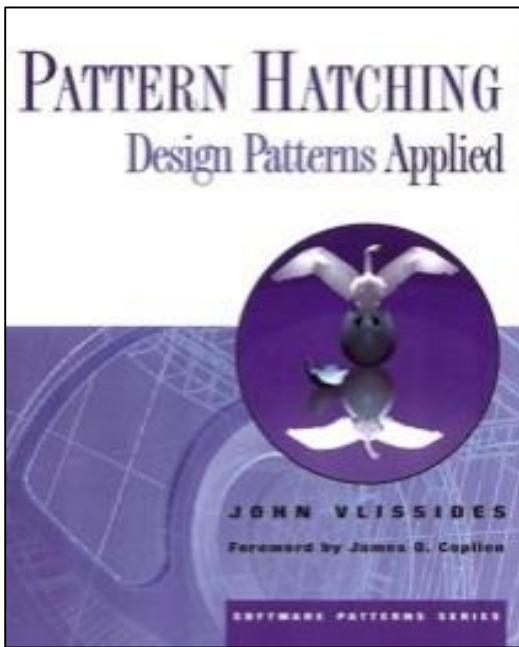
- The Gang-of-Four authors worked on their book for years
 - They selected design practices that could be recast as patterns, distilled the presentations, culled those deemed immature, etc.



See c2.com/cgi/wiki?HistoryOfPatterns for brief history of patterns

Brief History of the Gang-of-Four Pattern Book

- The Gang-of-Four authors worked on their book for years
 - They selected design practices that could be recast as patterns, distilled the presentations, culled those deemed immature, etc.
 - Some patterns that didn't make it into the GoF book appeared in John Vlissides later writings

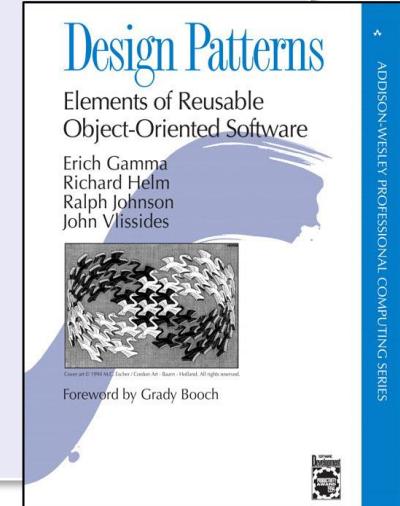


See c2.com/cgi/wiki?HistoryOfPatterns for brief history of patterns

Brief History of the Gang-of-Four Pattern Book

- The Gof book presents recurring solutions to common problems in software design in the form of 23 patterns

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

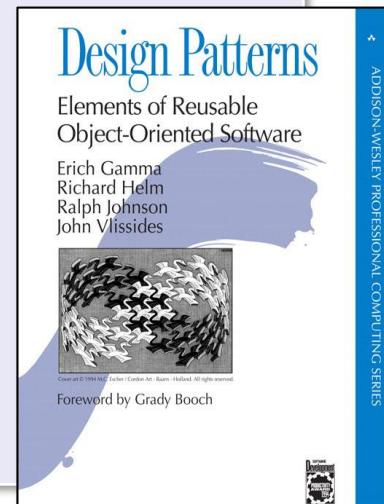


Brief History of the Gang-of-Four Pattern Book

Purpose: Reflects What the Pattern Does

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Scope: Domain Where Pattern Applies



The image shows the front cover of the book 'Design Patterns: Elements of Reusable Object-Oriented Software'. The title is at the top in a large blue serif font. Below it is a subtitle 'Elements of Reusable Object-Oriented Software'. The authors' names are listed: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. The cover features a black and white abstract graphic of a flock of birds in flight. At the bottom, there is a small note about the copyright and a foreword by Grady Booch.

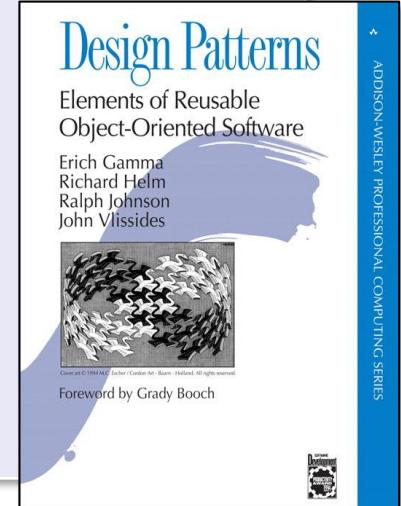
Brief History of the Gang-of-Four Pattern Book

Abstract the process of instantiating objects

Purpose: Reflects What the Pattern Does

Scope: Domain Where Pattern Applies

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



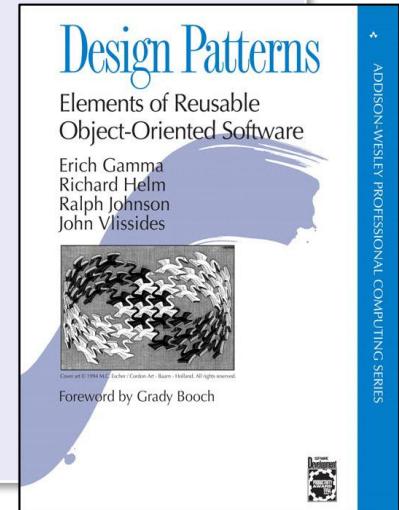
Brief History of the Gang-of-Four Pattern Book

Describe how classes & objects can be combined to form larger structures

Purpose: Reflects What the Pattern Does

Scope: Domain Where Pattern Applies

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



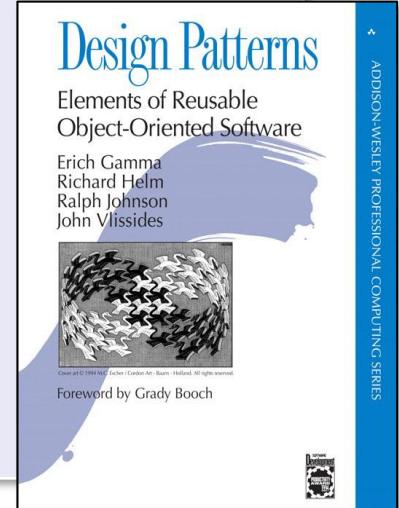
Brief History of the Gang-of-Four Pattern Book

Concerned with interactions between objects & distribution of responsibility

Purpose: Reflects What the Pattern Does

Scope: Domain Where Pattern Applies

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

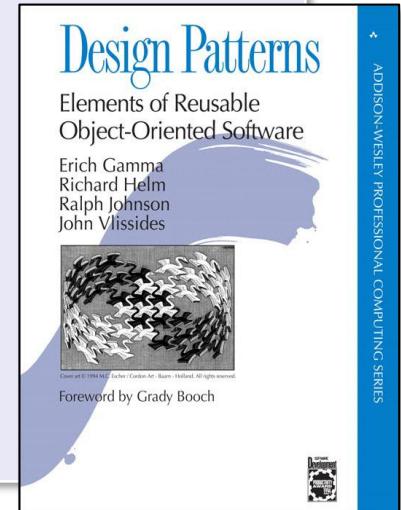


Brief History of the Gang-of-Four Pattern Book

Purpose: Reflects What the Pattern Does

Scope: Domain Where Pattern Applies

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

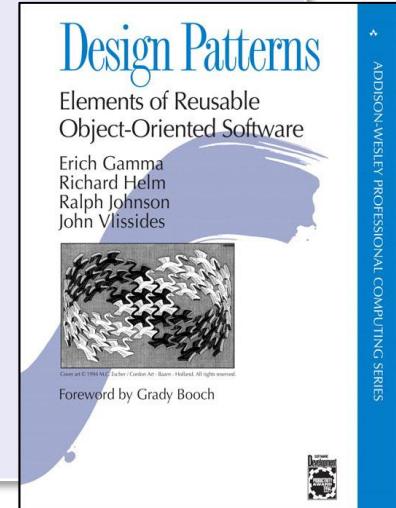
Brief History of the Gang-of-Four Pattern Book

Class patterns deal with relationships between classes & their subclasses

Purpose: Reflects What the Pattern Does

Scope: Domain Where Pattern Applies

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

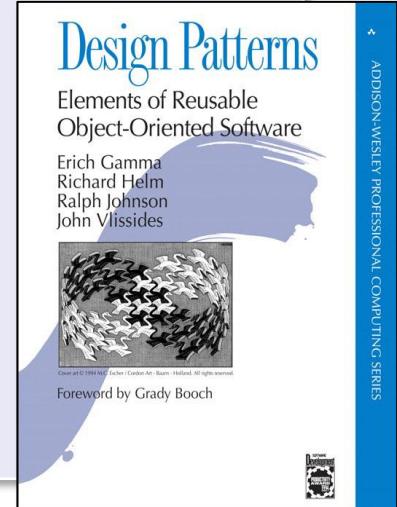
Brief History of the Gang-of-Four Pattern Book

Object patterns deal with object relationships that can be changed at run-time

Purpose: Reflects What the Pattern Does

Scope: Domain Where Pattern Applies

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Façade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Overview of Pattern Relationships

Overview of Pattern Relationships

- Stand-alone “pattern islands” are unusual in practice
 - Any substantial software design inevitably includes many patterns



Overview of Pattern Relationships

- Patterns are often related & are typically used together



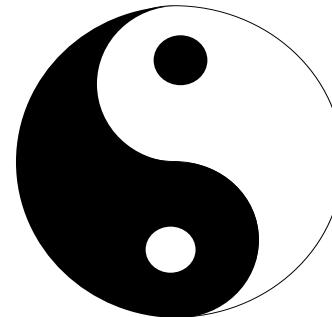
Overview of Pattern Relationships

- There are various types of pattern relationships

- **Pattern complements**

- One pattern provides missing ingredient needed by another

*Factory
Method*



*Disposal
Method*

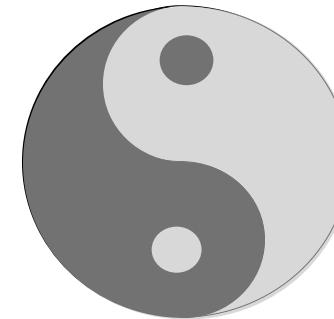
Overview of Pattern Relationships

- There are various types of pattern relationships

- **Pattern complements**

- One pattern provides missing ingredient needed by another

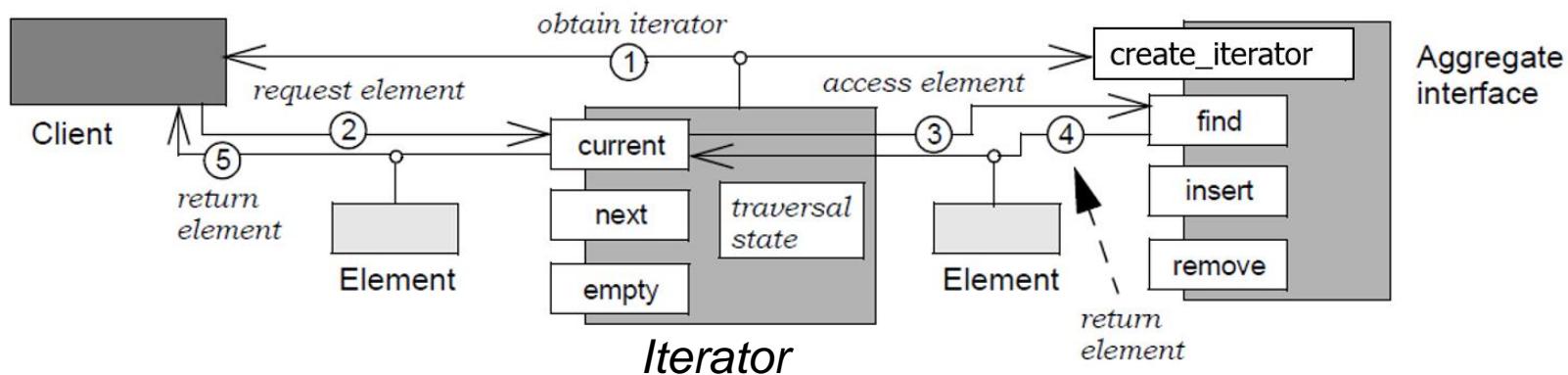
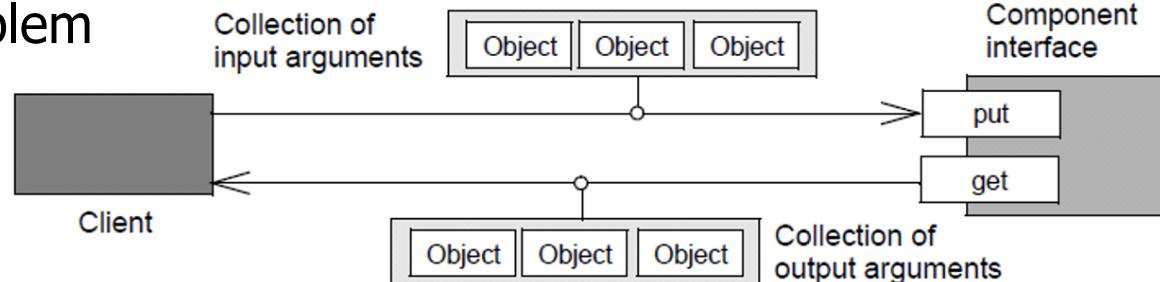
Factory Method



Disposal Method

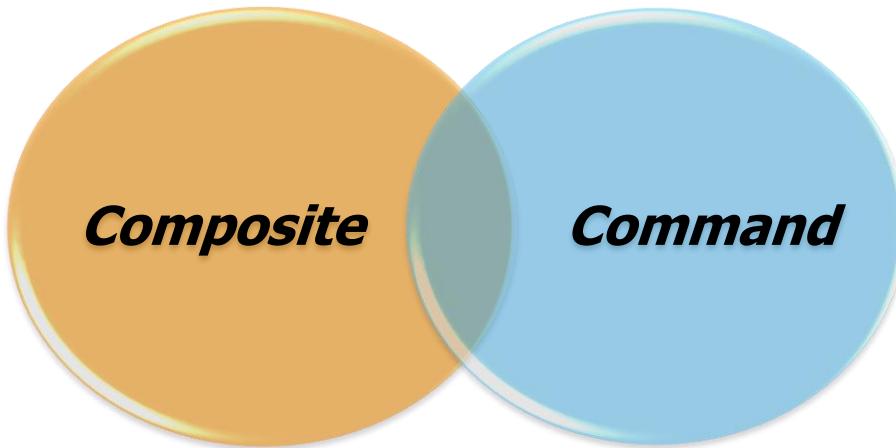
- One pattern contrasts w/another to provide an alternative solution to a related design problem

Batch Method



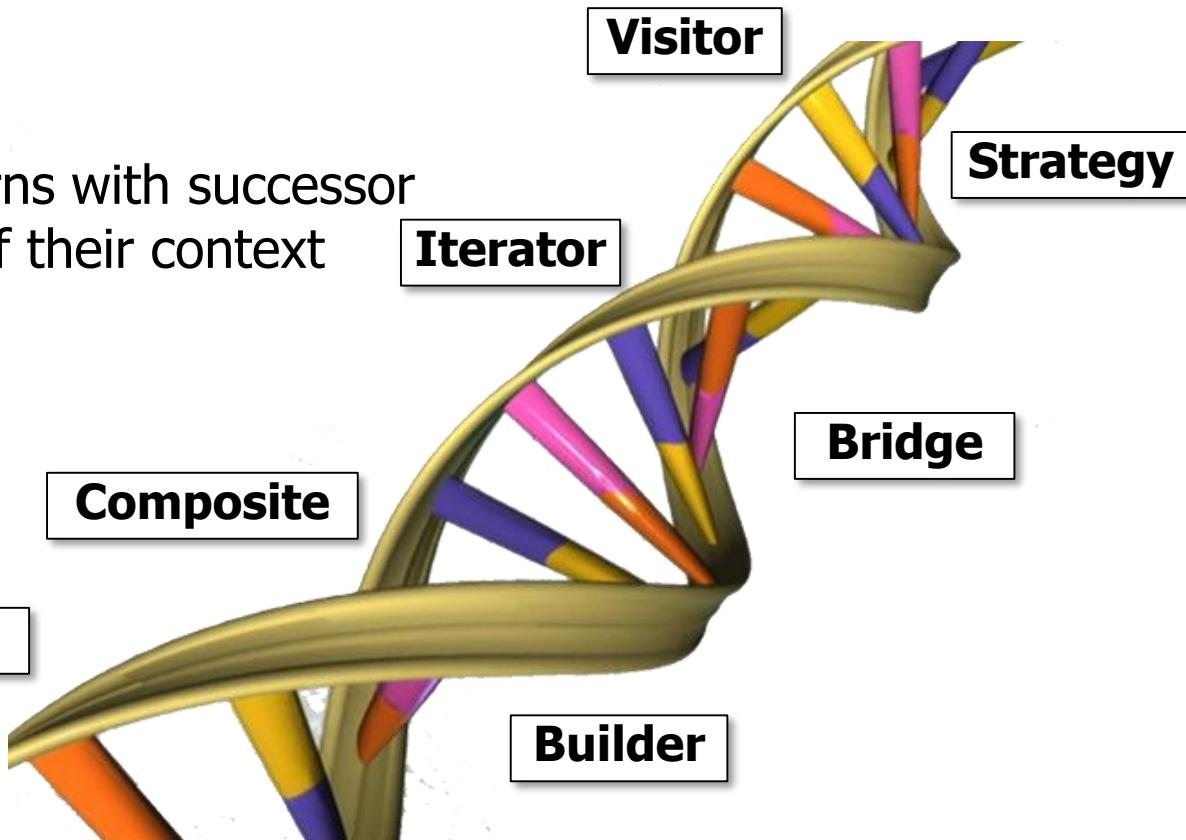
Overview of Pattern Relationships

- There are various types of pattern relationships
 - **Pattern complements**
 - **Pattern compounds**
 - Capture recurring subcommunities of patterns that can be treated as a single decision in response to a recurring design problem



Overview of Pattern Relationships

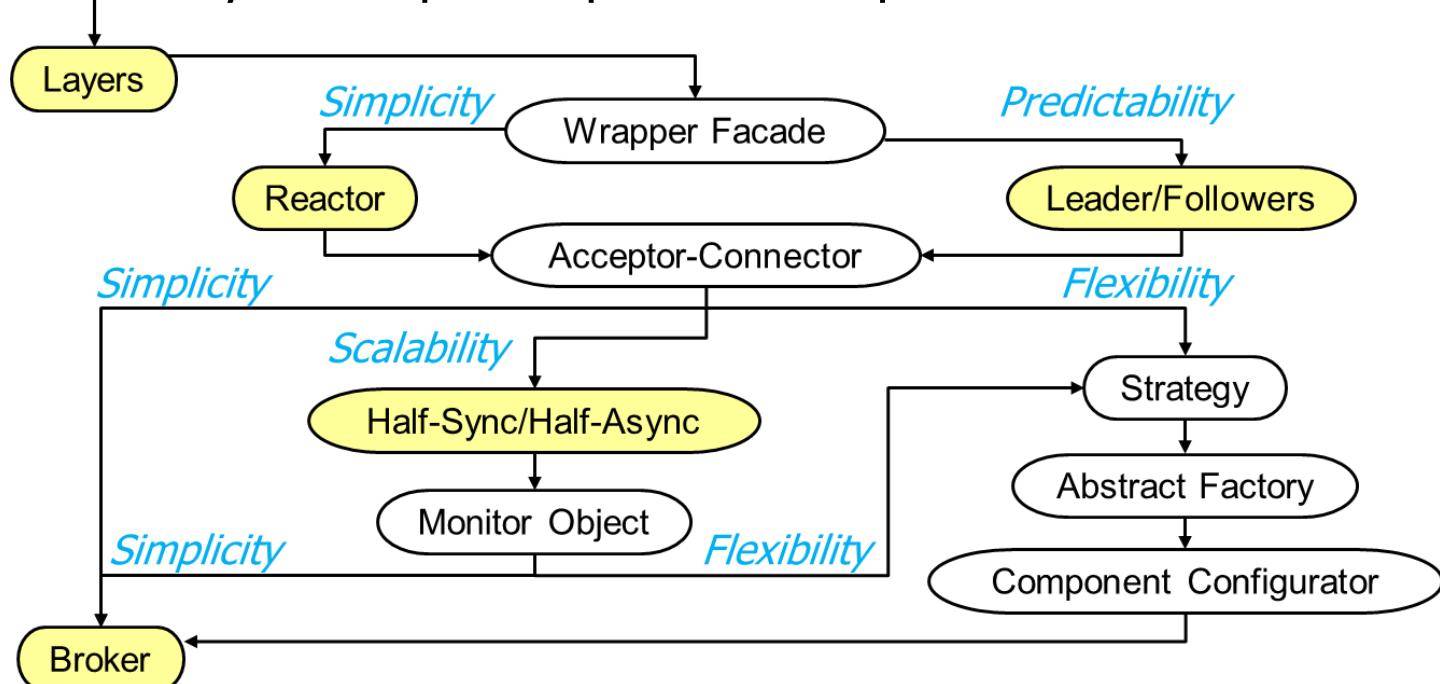
- There are various types of pattern relationships
 - **Pattern complements**
 - **Pattern compounds**
 - **Pattern sequences**
 - Join predecessor patterns with successor patterns to form part of their context



We'll present a sequence of GoF patterns later in the course

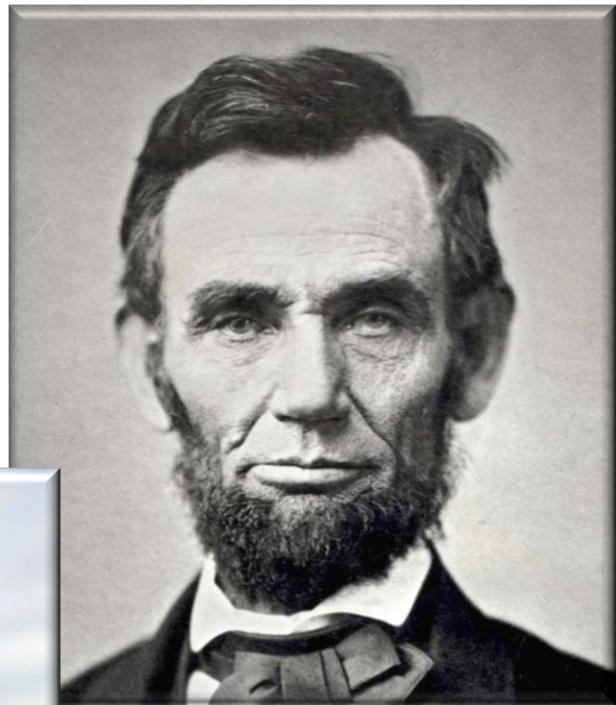
Overview of Pattern Relationships

- There are various types of pattern relationships
 - **Pattern complements**
 - **Pattern compounds**
 - **Pattern sequences**
 - **Pattern languages**
 - Networks of related patterns that define a process for the orderly resolution of key development problems in particular domains



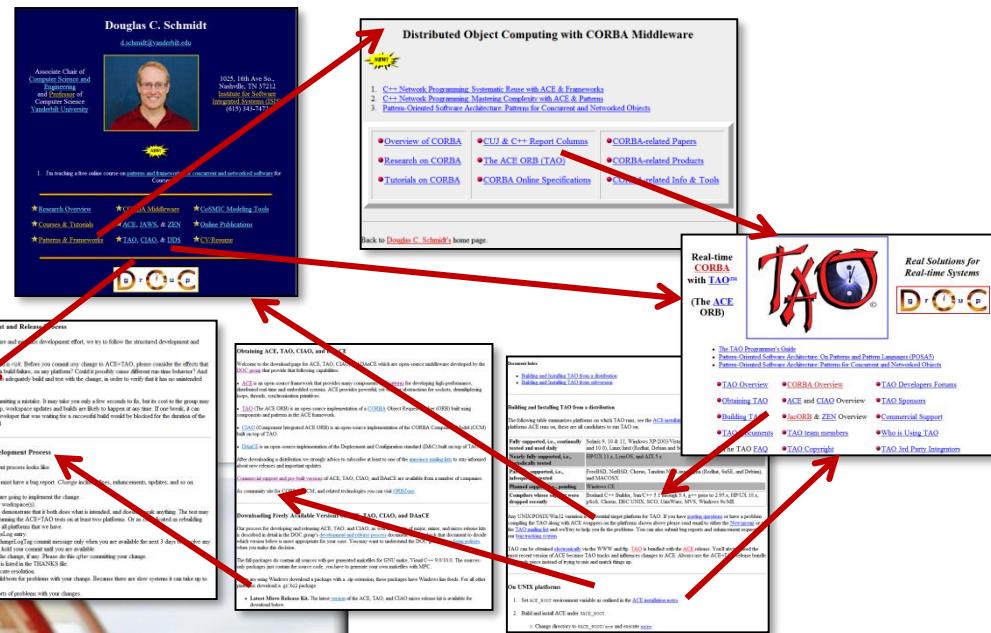
Comparing Patterns, Sequences, & Languages

- A pattern is to a pattern sequence as a photograph is to a traditional film
 - i.e., a pattern sequence is a linearized presentation of related patterns



Comparing Patterns, Sequences, & Languages

- A pattern sequence is to a pattern language as a traditional film is to a “hyperlink cinema” film or a hypertext document
 - i.e., a network of patterns that can be traversed & applied in many different ways



Putting All the Pieces Together

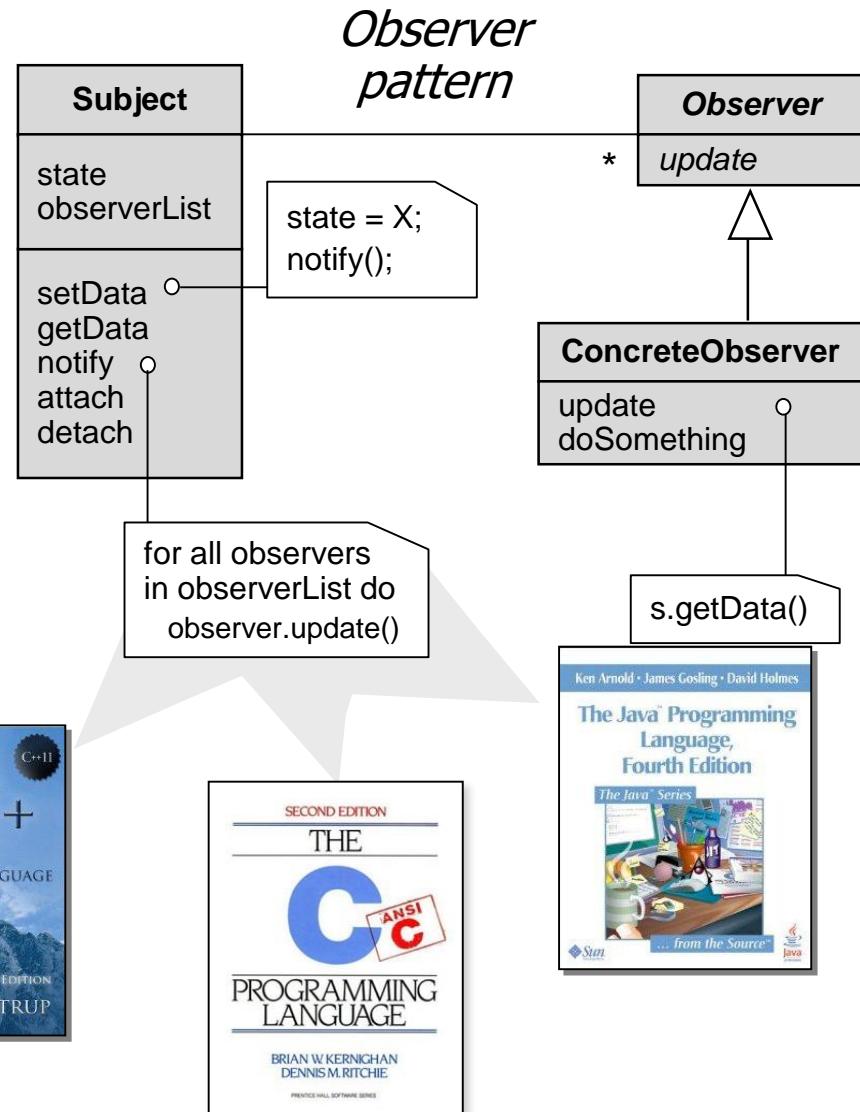
Putting All the Pieces Together

- Achieving mastery of software development requires continuous repetition, practice, & mentoring from experts



Putting All the Pieces Together

- Patterns codify software expertise & support design at a more abstract level than code
 - Emphasize design *qua* design, not (obscure) language features
 - e.g., the *Observer* pattern is implemented in many different programming languages



Patterns often equated with OO languages, but can apply to non-OO languages

Putting All the Pieces Together

- Patterns codify software expertise & support design at a more abstract level than code
 - Emphasize design *qua* design, not (obscure) language features
 - Treat class/object interactions as a cohesive conceptual unit
 - e.g., form building blocks for more powerful pattern relationships

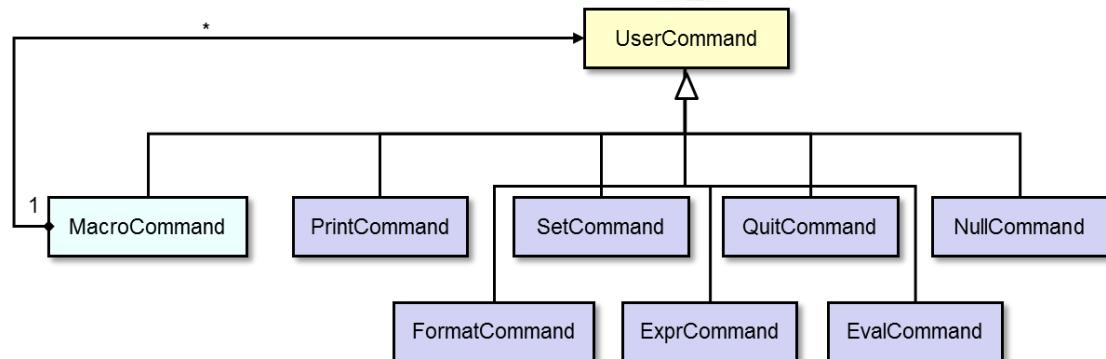
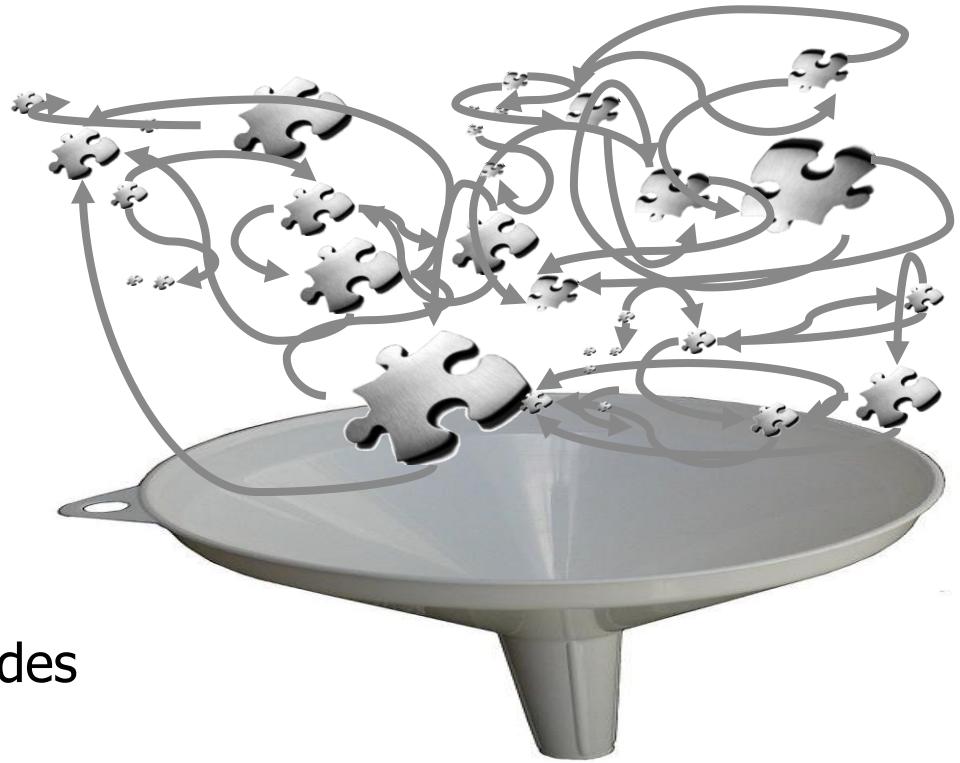
Adapter **Strategy**



Observer **Proxy**

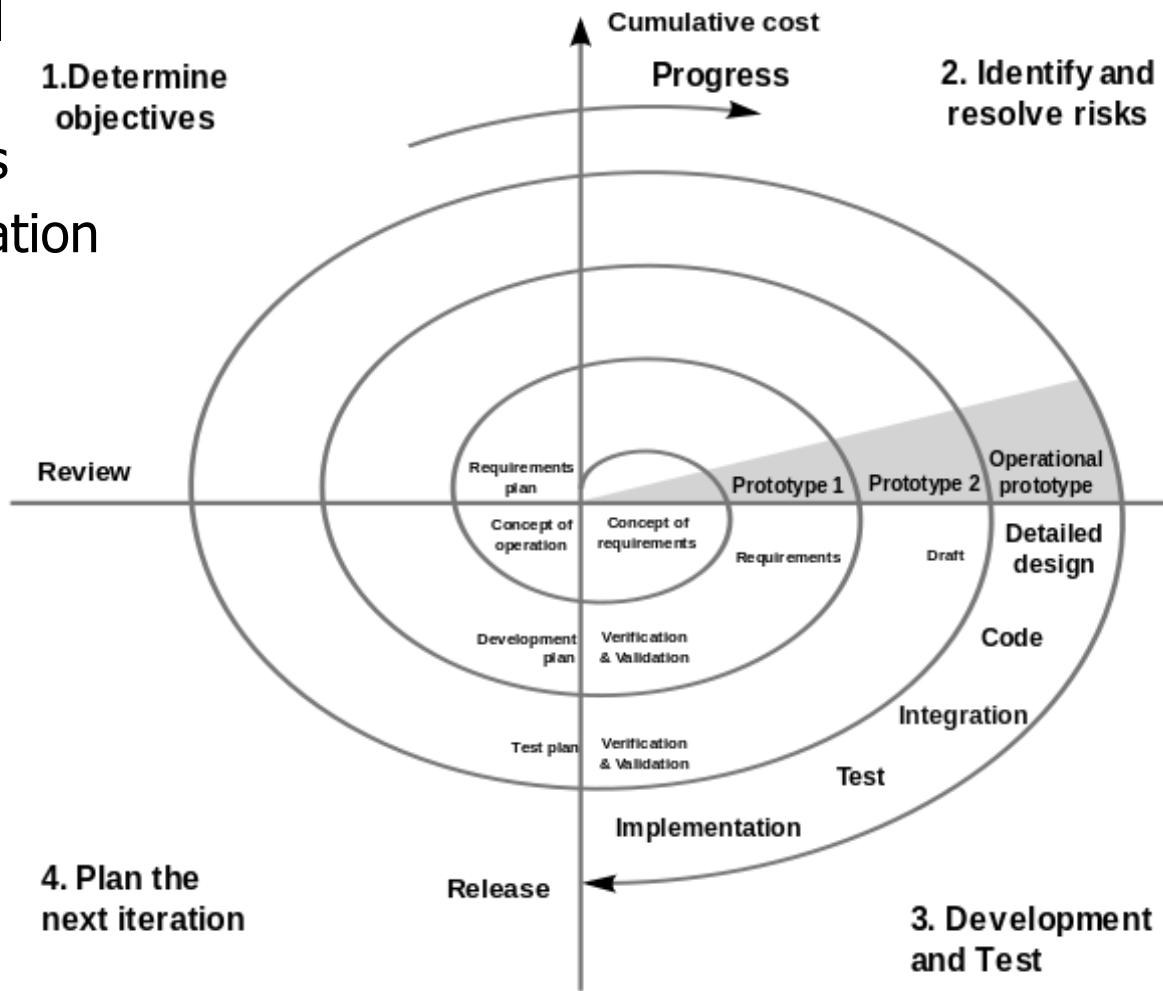
Putting All the Pieces Together

- Patterns codify software expertise & support design at a more abstract level than code
 - Emphasize design *qua* design, not (obscure) language features
 - Treat class/object interactions as a cohesive conceptual unit
 - Provide ideal targets for design & implementation refactoring
 - e.g., adapters & (wrapper) facades



Putting All the Pieces Together

- Patterns can be applied in all software lifecycle phases
 - Analysis, design, & reviews
 - Implementation & optimization
 - Testing & documentation
 - Reuse & refactoring



Putting All the Pieces Together

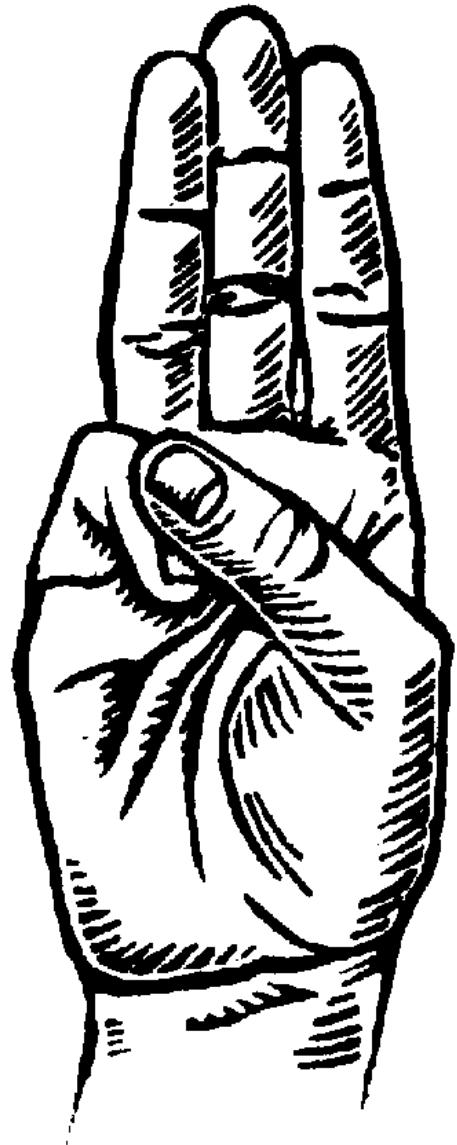
- Seek generality, but don't brand everything as a pattern



Putting All the Pieces Together

- Articulate specific benefits & demonstrate general applicability
 - e.g., find three different existing examples from code other than yours!

Rule of Three



**End of Overview
of Pattern Concepts**