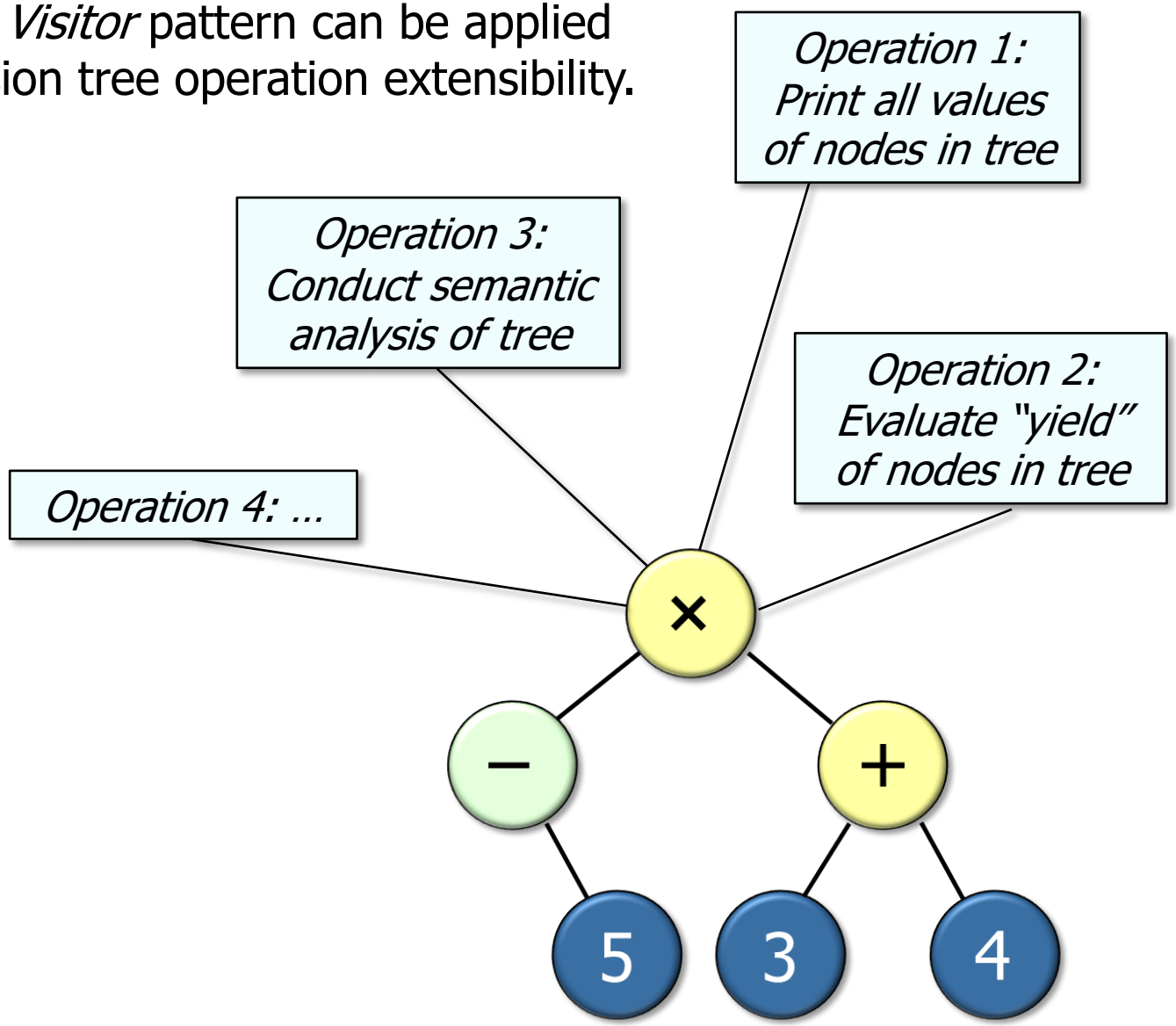
The Visitor Pattern

Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Visitor* pattern can be applied to enhance expression tree operation extensibility.

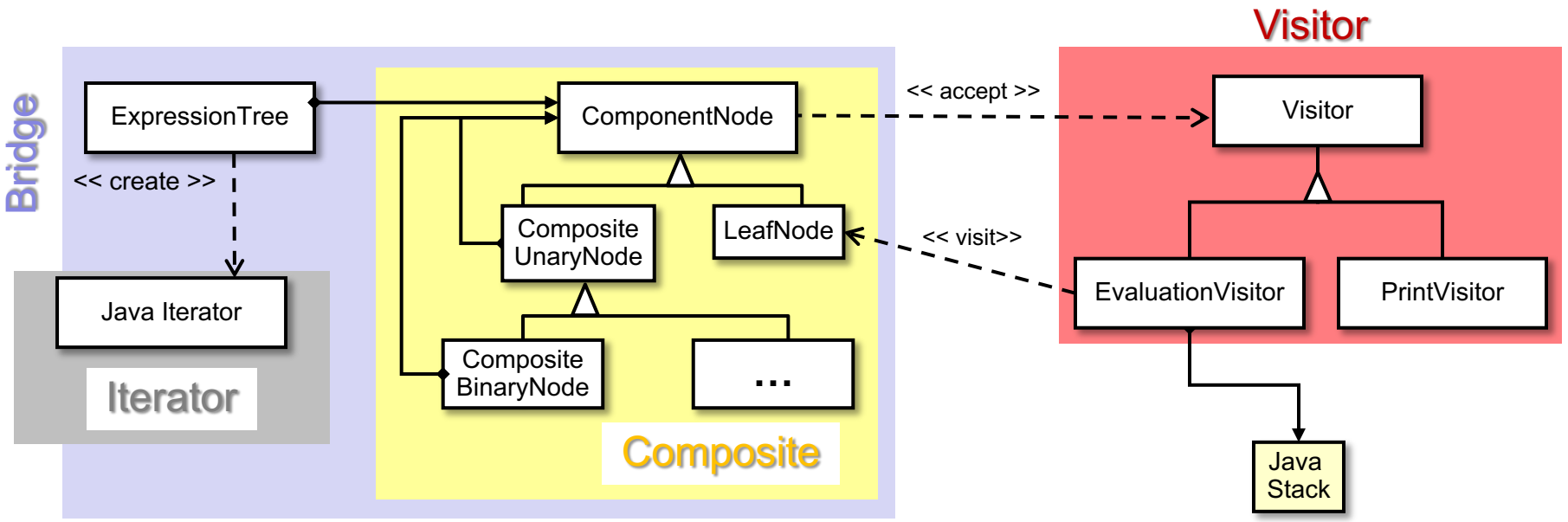


Douglas C. Schmidt

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

A Pattern for Applying Operations on a Composite

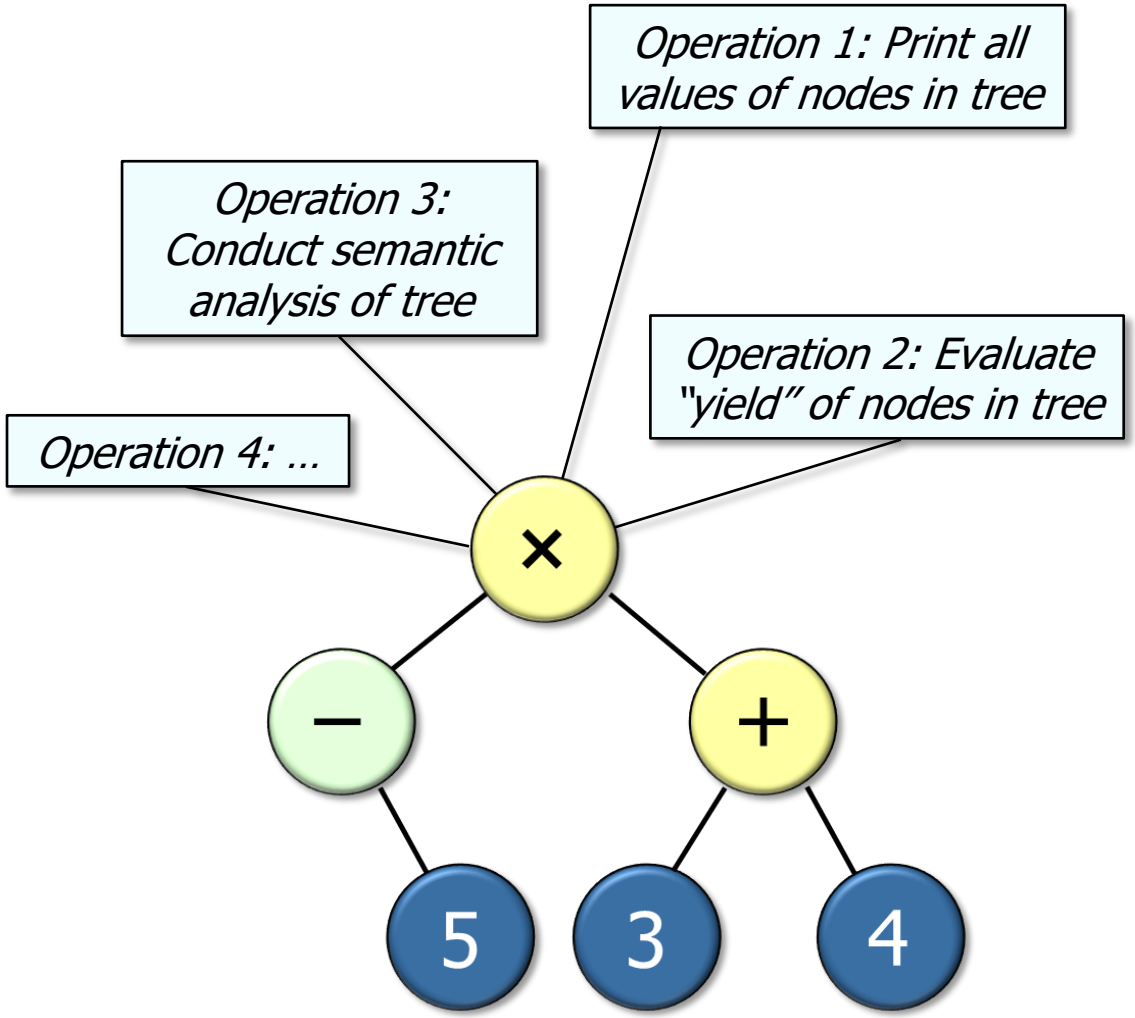
***Purpose**: Perform an extensible set of operations on an expression tree without requiring any changes to the tree itself.*



Visitor decouples expression tree structure from operations performed on it.

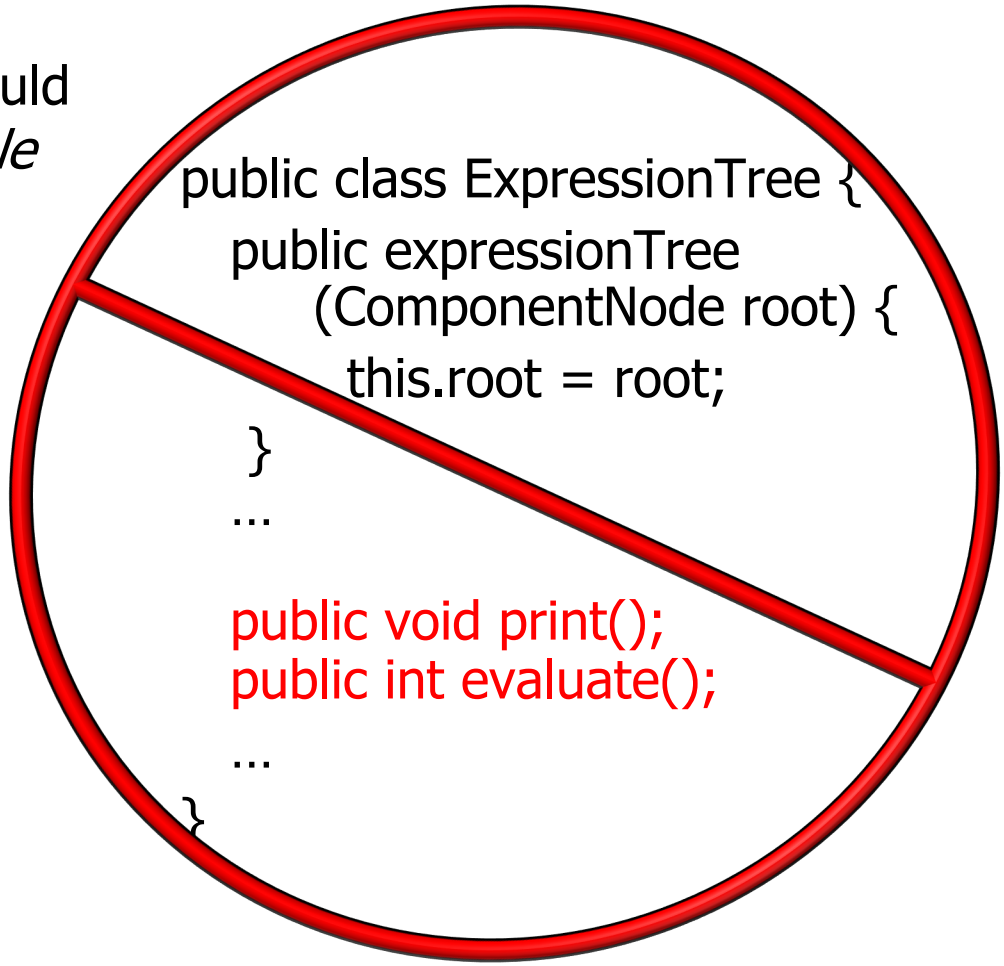
Context: OO Expression Tree Processing App

- Adding new operations to an expression tree should require no changes to the tree's structure and implementation.



Problem: Non-Extensible Tree Operations

- Hard-coding operations in **`ExpressionTree`** or in **`ComponentNode`** subclasses limits extensibility.
  - e.g., adding new operations would violate the *Open/Closed Principle* since the **`ExpressionTree`** class API would change.



See en.wikipedia.org/wiki/Open/closed_principle

Problem: Non-Extensible Tree Operations

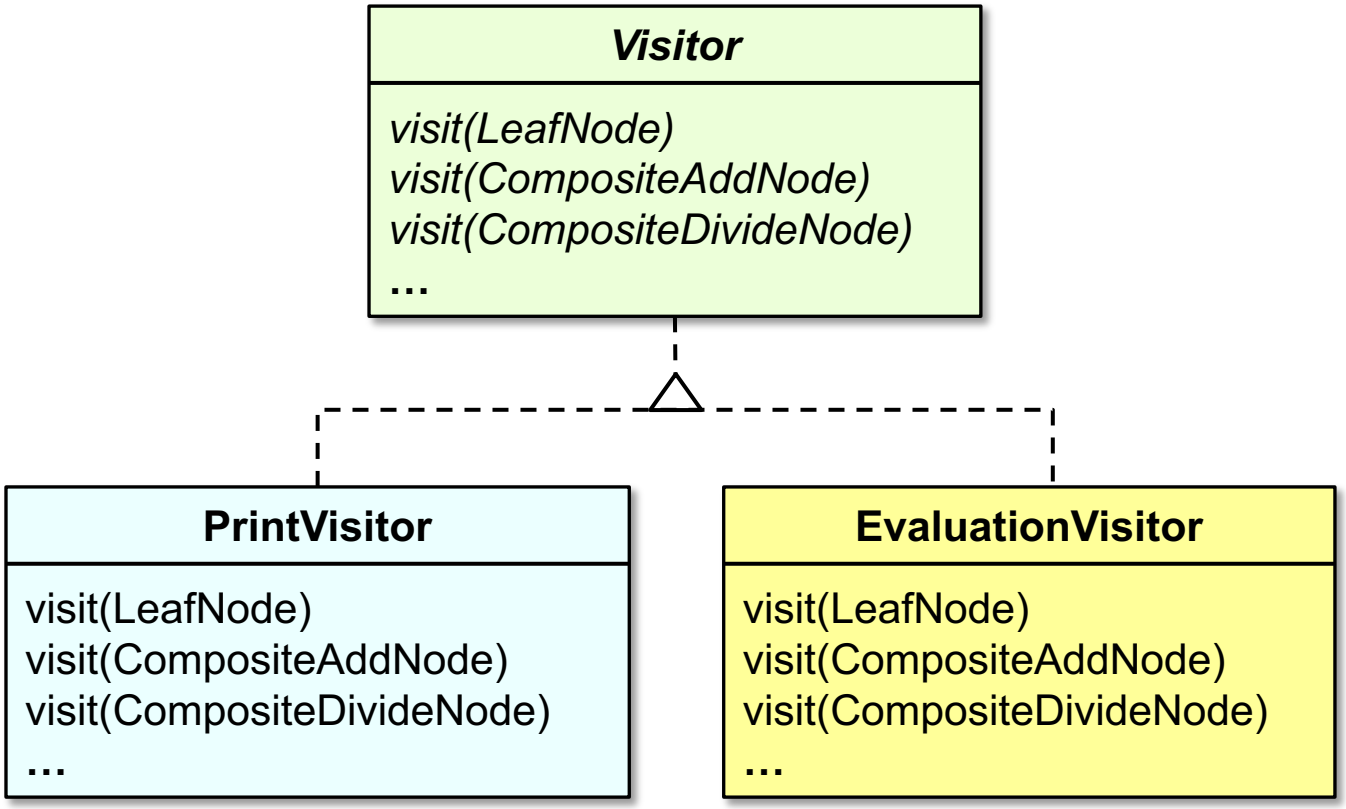
- Hard-coding downcasts to access ExpressionTree nodes also limits extensibility.

```
ExpressionTree exprTree = ...;

for(Iterator<ExpressionTree> it = exprTree.iterator();
    it.hasNext();
    ) {
   ExpressionTree node = it.next();
   if(node.getRoot() instanceof LeafNode)
     System.out.print((int)node.getItem() + " ");
   else
     System.out.print((char)node.getItem() + " ");
}
```

Coding like this will cause trouble at some point…

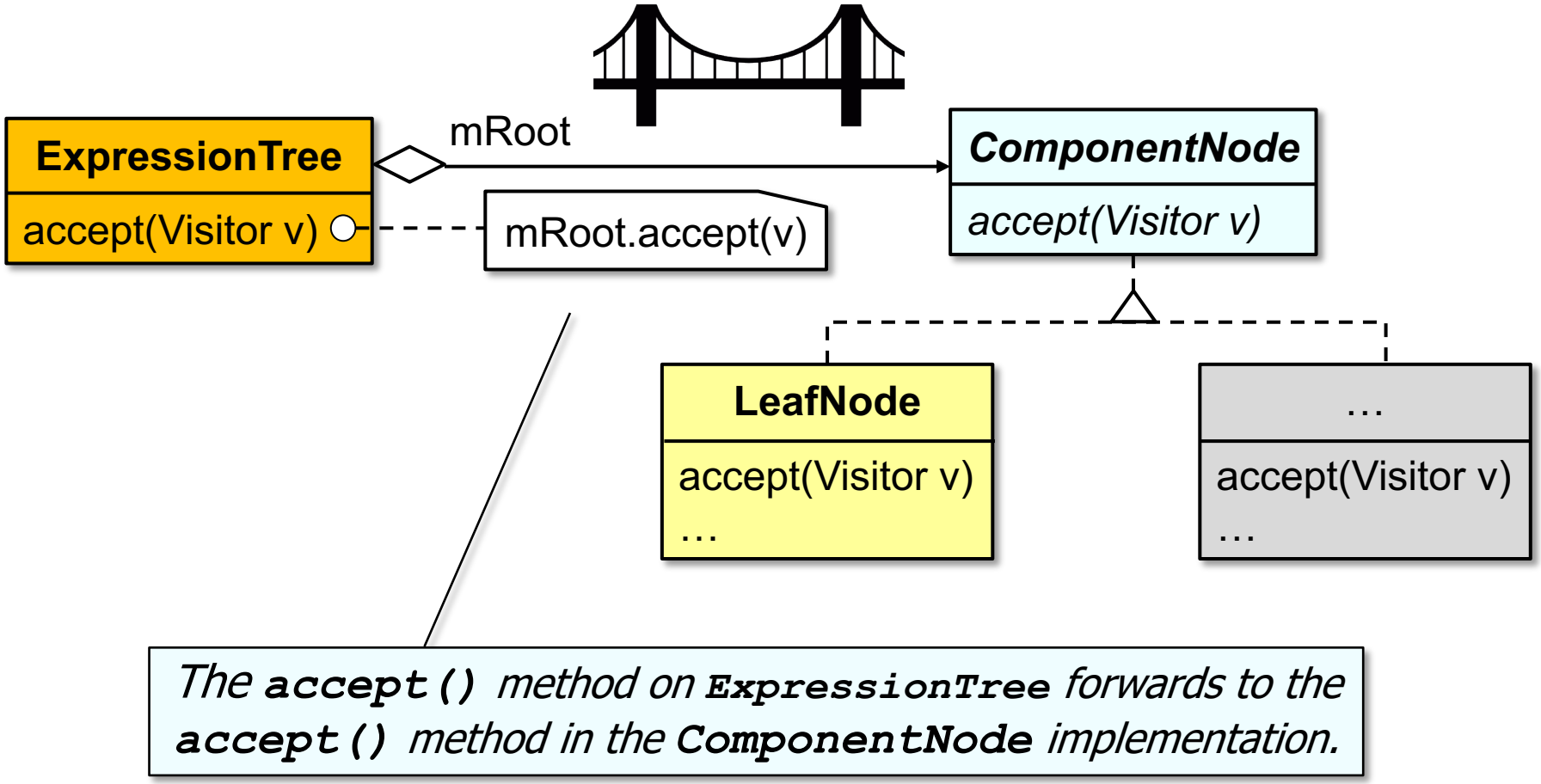
Solution: Decouple Operations From Structure

- Create a hierarchy of visitors that define **`visit()`** methods to perform on each expression tree node implementation.



Solution: Decouple Operations From Structure

- Define an **`accept()`** method in the **`ExpressionTree`** class API that is passed an instance of a visitor implementation.



Solution: Decouple Operations From Structure

- During an iteration over the expression tree call **`accept()`** on each node and pass in the visitor instance, e.g.,

```
Visitor printVisitor = visitorFactory.makeVisitor("print");
```

Use a factory method to create a print visitor

```
ExpressionTree tree = makeExpressionTree("-5 * (3 + 4)");
```

```
for(Iterator<ExpressionTree> it = tree.iterator("post-order");
    it.hasNext();)
  it.next().accept(printVisitor);
```

Solution: Decouple Operations From Structure

- During an iteration over the expression tree call **`accept()`** on each node and pass in the visitor instance, e.g.,

```
Visitor printVisitor = visitorFactory.makeVisitor("print");


ExpressionTree tree = makeExpressionTree("-5 * (3 + 4)");
```

Call a factory method to create an expression tree

```
for(Iterator<ExpressionTree> it = tree.iterator("post-order");
    it.hasNext();)
  it.next().accept(printVisitor);
```

Solution: Decouple Operations From Structure

- During an iteration over the expression tree call **`accept()`** on each node and pass in the visitor instance, e.g.,

```
Visitor printVisitor = visitorFactory.makeVisitor("print");


ExpressionTree tree = makeExpressionTree("-5 * (3 + 4)");


for(Iterator<ExpressionTree> it = tree.iterator("post-order");
    it.hasNext();)
  it.next().accept(printVisitor);
```

*The **`accept()`** method on **`ExpressionTree`** forwards to the **`accept()`** method in the **`ComponentNode`** implementation.*

Solution: Decouple Operations From Structure

- Have **`accept()`** call back to the **`visitor.visit()`** method, passing in the corresponding node in the expression tree to perform the operation, e.g.,

```
public class LeafNode implements ComponentNode {
  public void accept(Visitor visitor)
    visitor.visit(this);
  }
...
```

This indirection avoids hard-coding operations into expression tree nodes.

Solution: Decouple Operations From Structure

- Have **`accept()`** call back to the **`visitor.visit()`** method, passing in the corresponding node in the expression tree to perform the operation, e.g.,

```
public class LeafNode implements ComponentNode {
  public void accept(Visitor visitor)
    visitor.visit(this);
  }
...
```

Method overloading by the **`ComponentNode`** *subclass is "static polymorphism" that eliminates the need for ugly downcasts.*

Visitor

visit(LeafNode)
visit(CompositeAddNode)
visit(CompositeDivideNode)
...

See mrbool.com/static-polymorphism-in-java/29706

Visitor Interface Overview

- Specifies an extensible set of operations that can be performed on each implementation of **`ComponentNode`** in an expression tree

Interface methods

```
void visit(LeafNode)
void visit(CompositeNegateNode)
void visit(CompositeAddNode)
void visit(CompositeSubtractNode)
void visit(CompositeDivideNode)
void visit(CompositeMultiplyNode)
```

Visitor Interface Overview

- Specifies an extensible set of operations that can be performed on each implementation of **`ComponentNode`** in an expression tree

Interface methods

An overloaded visit() method is defined by each implementation of `ComponentNode`.

```
void  visit(LeafNode)
void  visit(CompositeNegateNode)
void  visit(CompositeAddNode)
void  visit(CompositeSubtractNode)
void  visit(CompositeDivideNode)
void  visit(CompositeMultiplyNode)
```

Visitor Interface Overview

- Specifies an extensible set of operations that can be performed on each implementation of **`ComponentNode`** in an expression tree

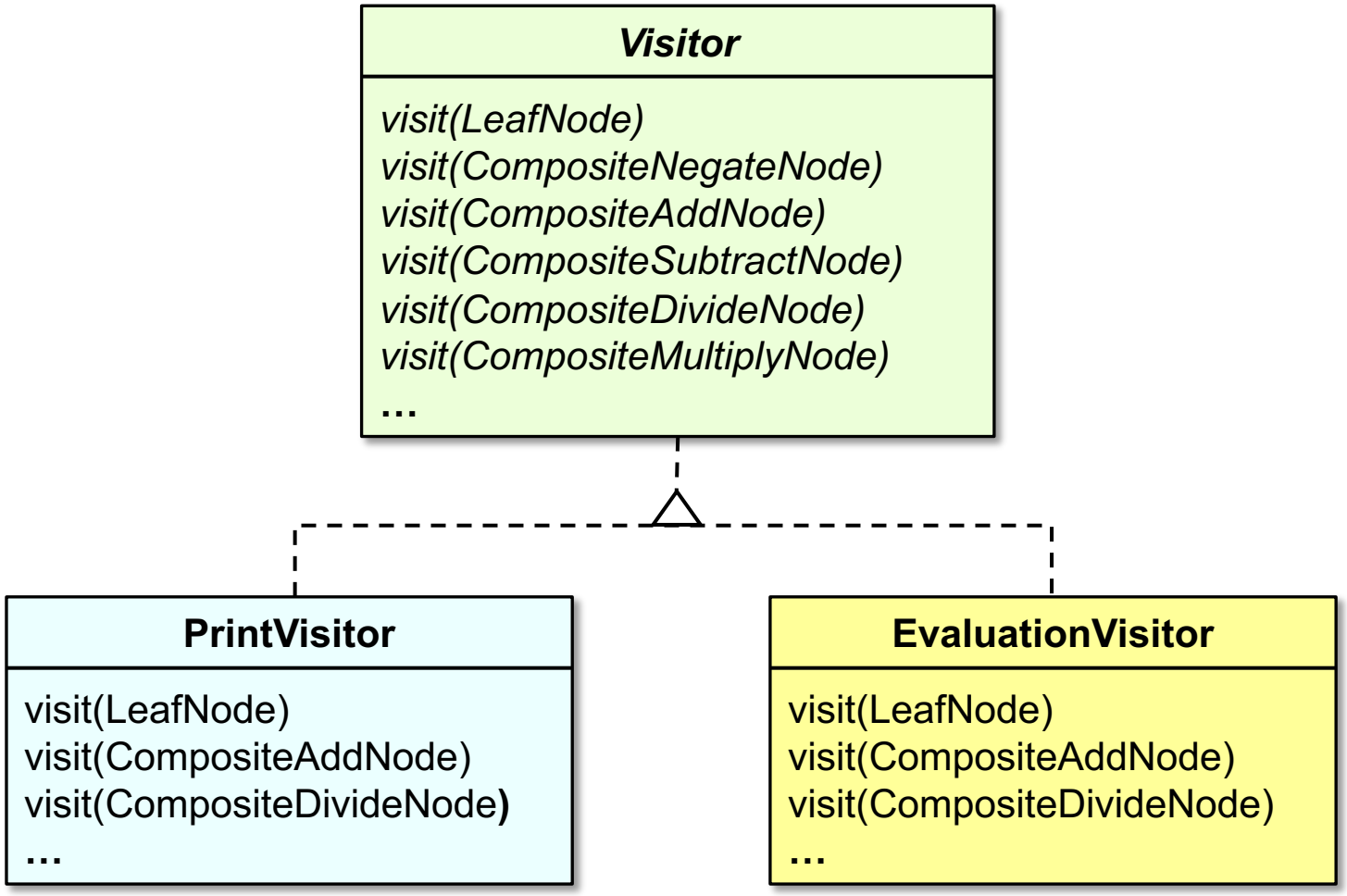
Interface methods

```
void visit(LeafNode)
void visit(CompositeNegateNode)
void visit(CompositeAddNode)
void visit(CompositeSubtractNode)
void visit(CompositeDivideNode)
void visit(CompositeMultiplyNode)
```

- **Commonality**: provides a common set of **`visit()`** methods, one for each implementation of **`ComponentNode`**
- **Variability**: implementations of this interface define specific behaviors for different types of visitors

Visitor Implementation Hierarchy Overview

- A class hierarchy that defines operations performed on implementations of **`ComponentNode`** in an expression tree



`Visitor` *implementations* define operations rather than the **`ExpressionTree`** API.

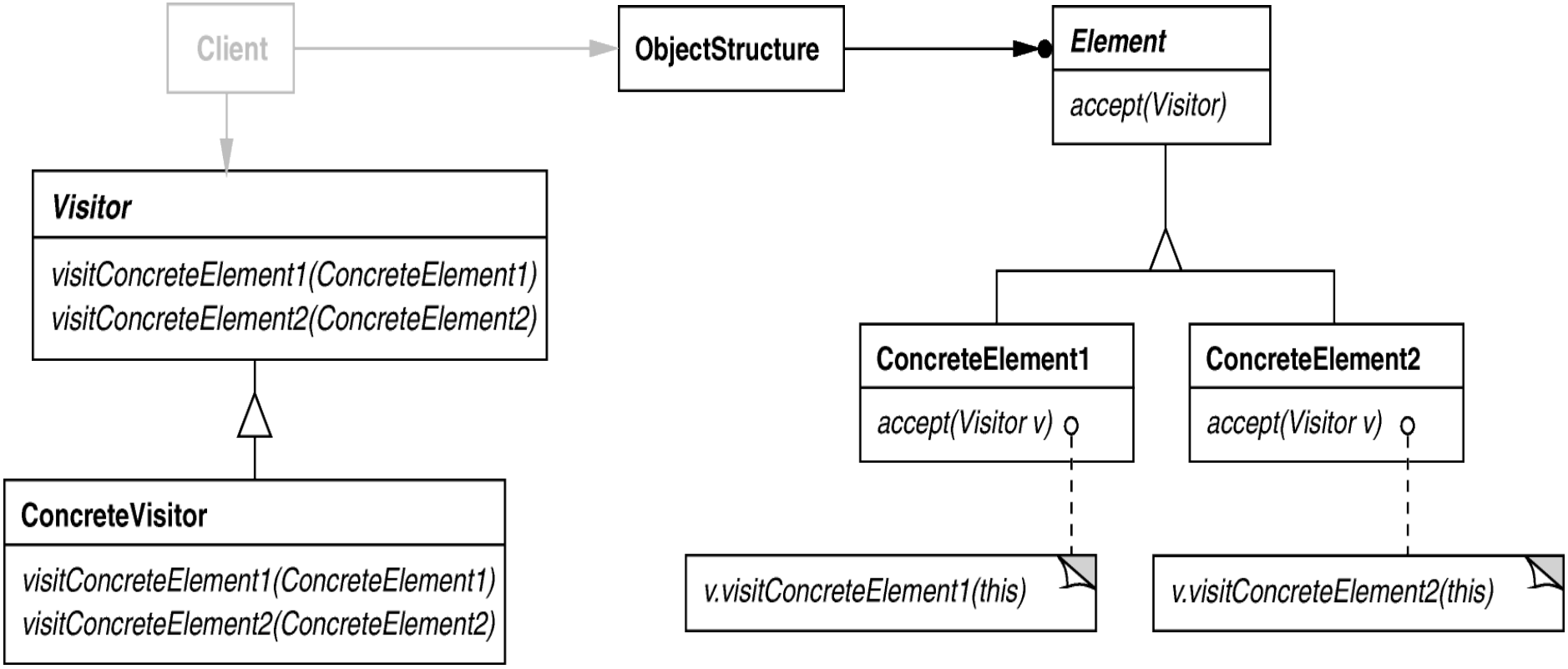
The Visitor Pattern

Structure and Functionality

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Visitor* pattern can be applied to enhance expression tree operation extensibility.
- Understand the *Visitor* pattern.



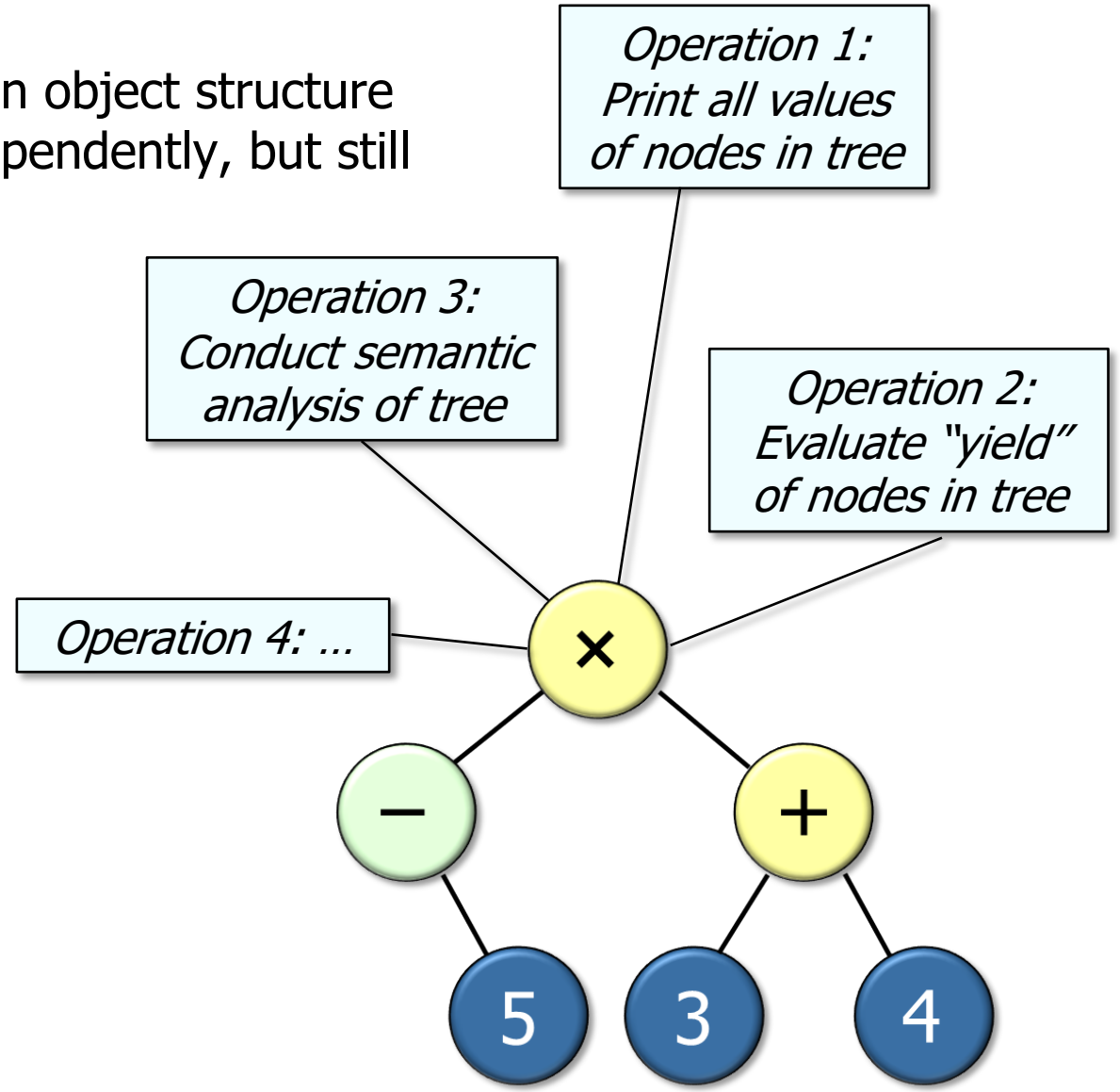
Visitor is one of the most complicated GoF patterns (along with *State*).

Douglas C. Schmidt

Structure and Functionality of the Visitor Pattern

Intent

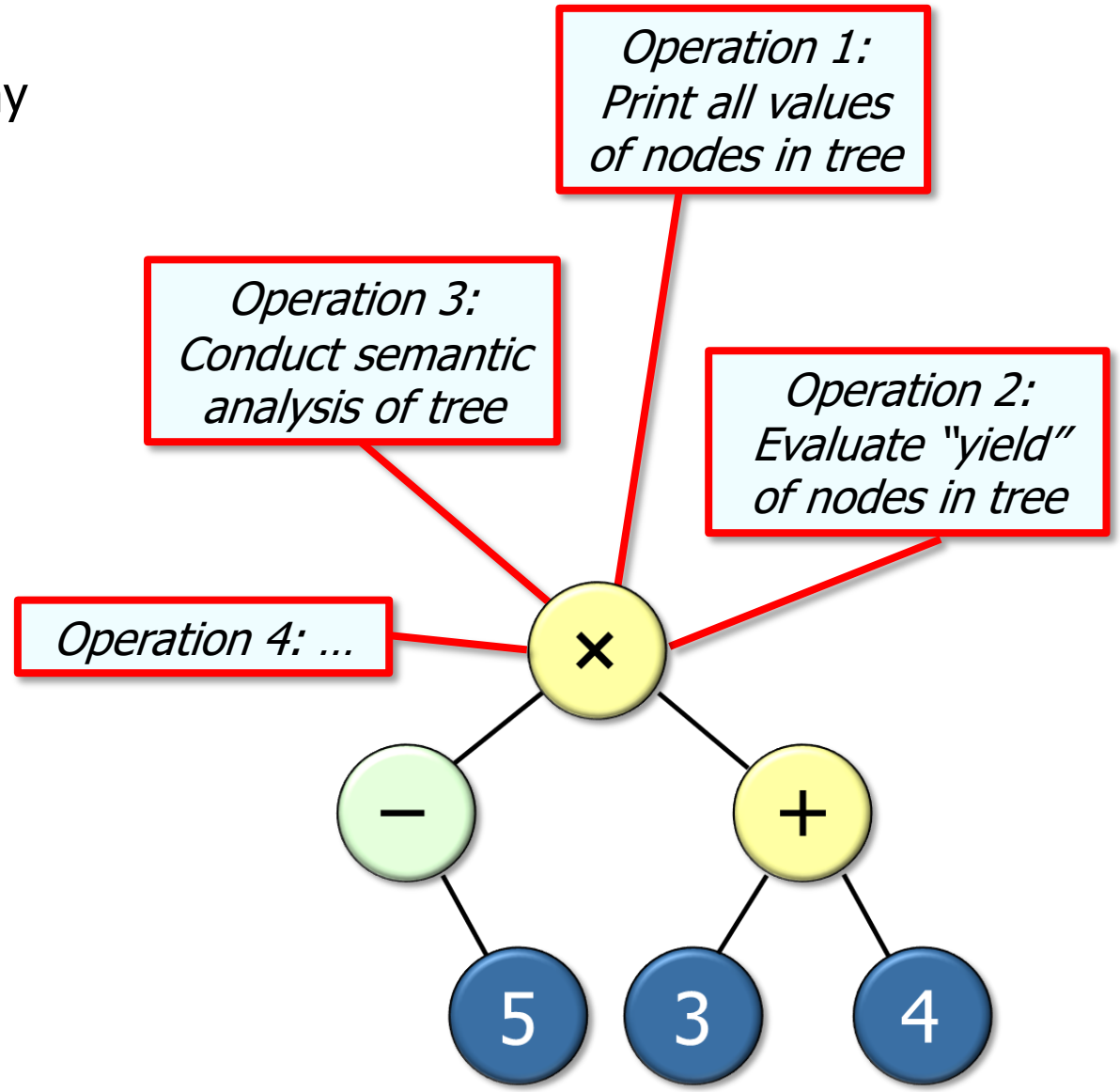
- Centralize operations on an object structure so that they can vary independently, but still behave polymorphically



See en.wikipedia.org/wiki/Visitor_pattern

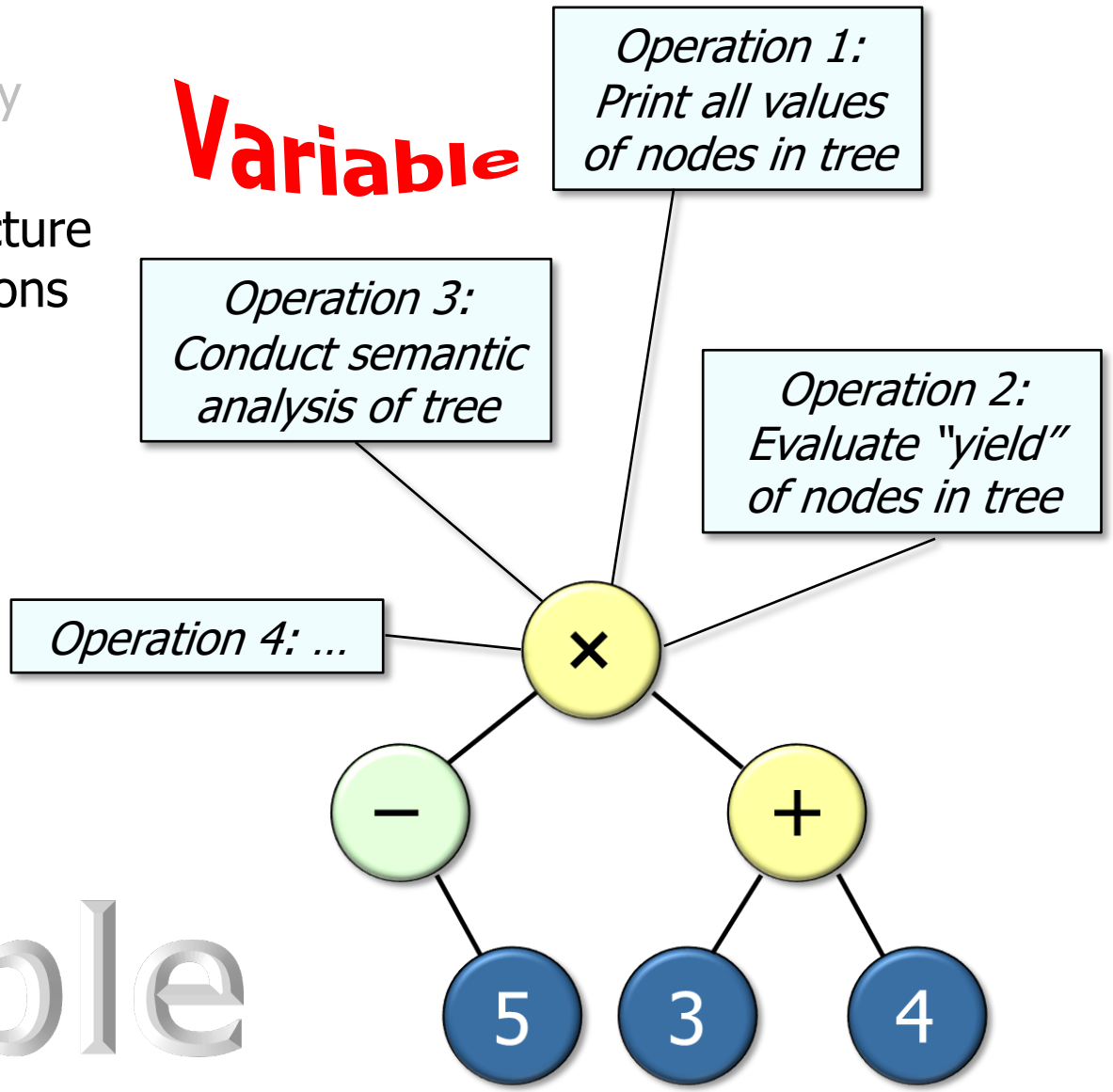
Applicability

- When classes involve many unrelated operations



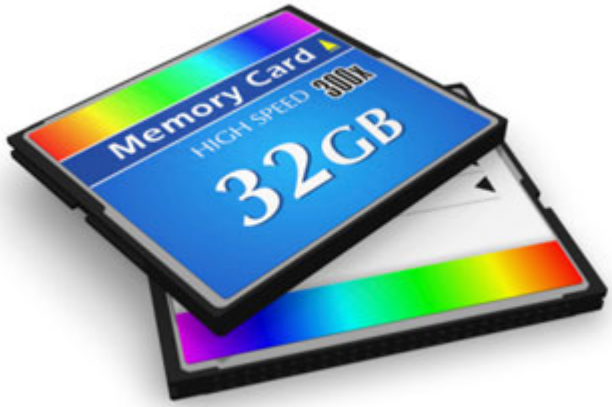
Applicability

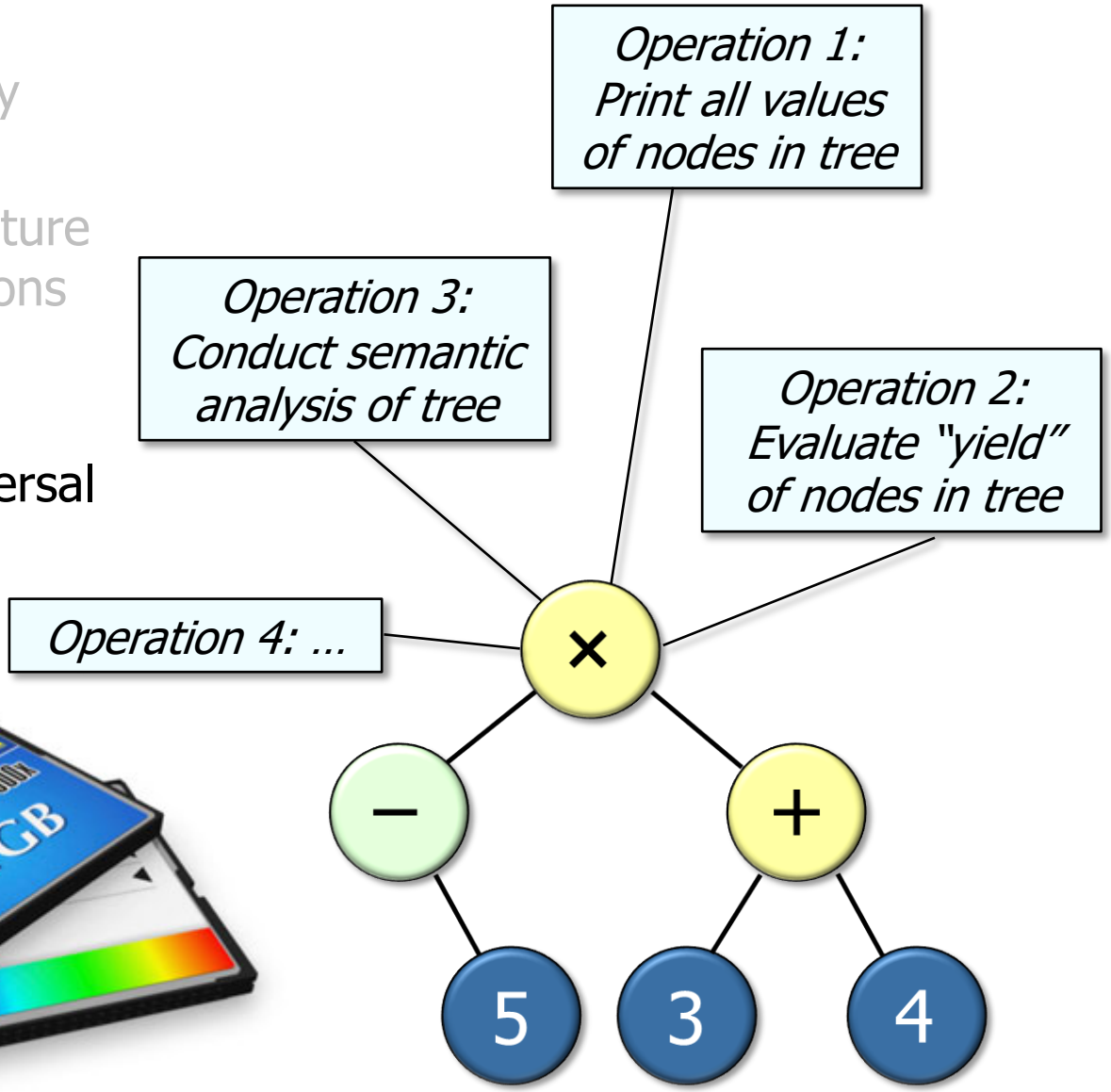
- When classes involve many unrelated operations
- Class relationships in structure rarely change, but operations on them *do* change



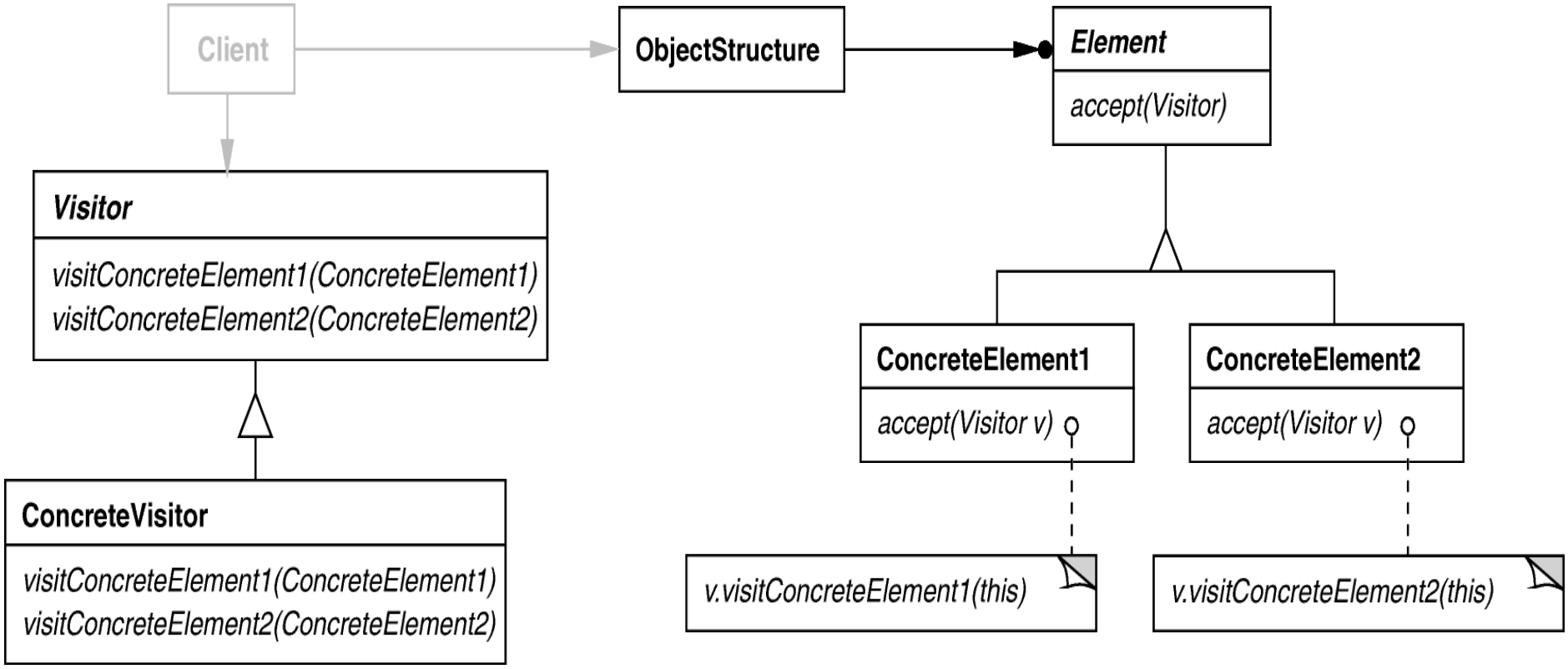
Applicability

- When classes involve many unrelated operations
- Class relationships in structure rarely change, but operations on them *do* change
- Algorithms keep the state that's updated during traversal

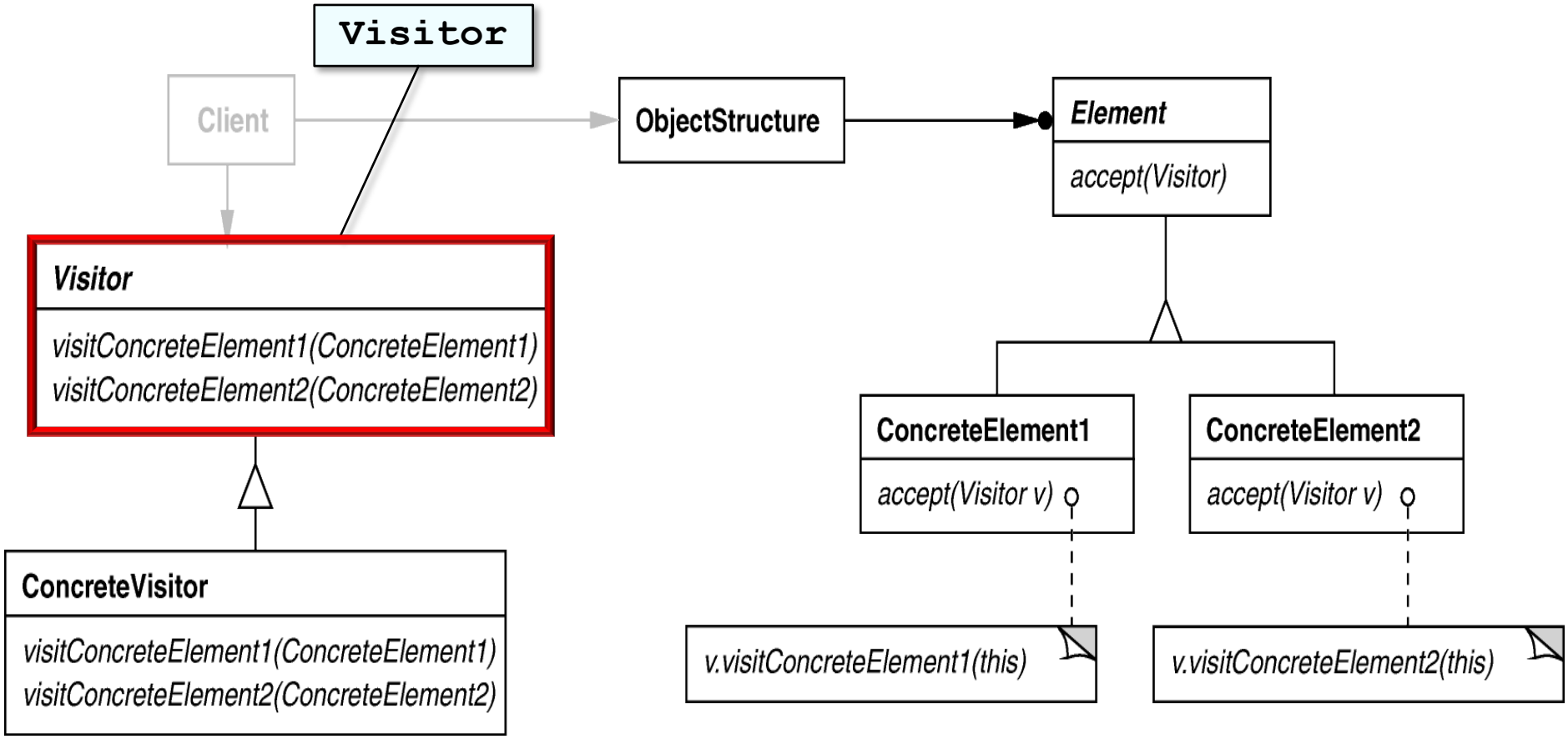




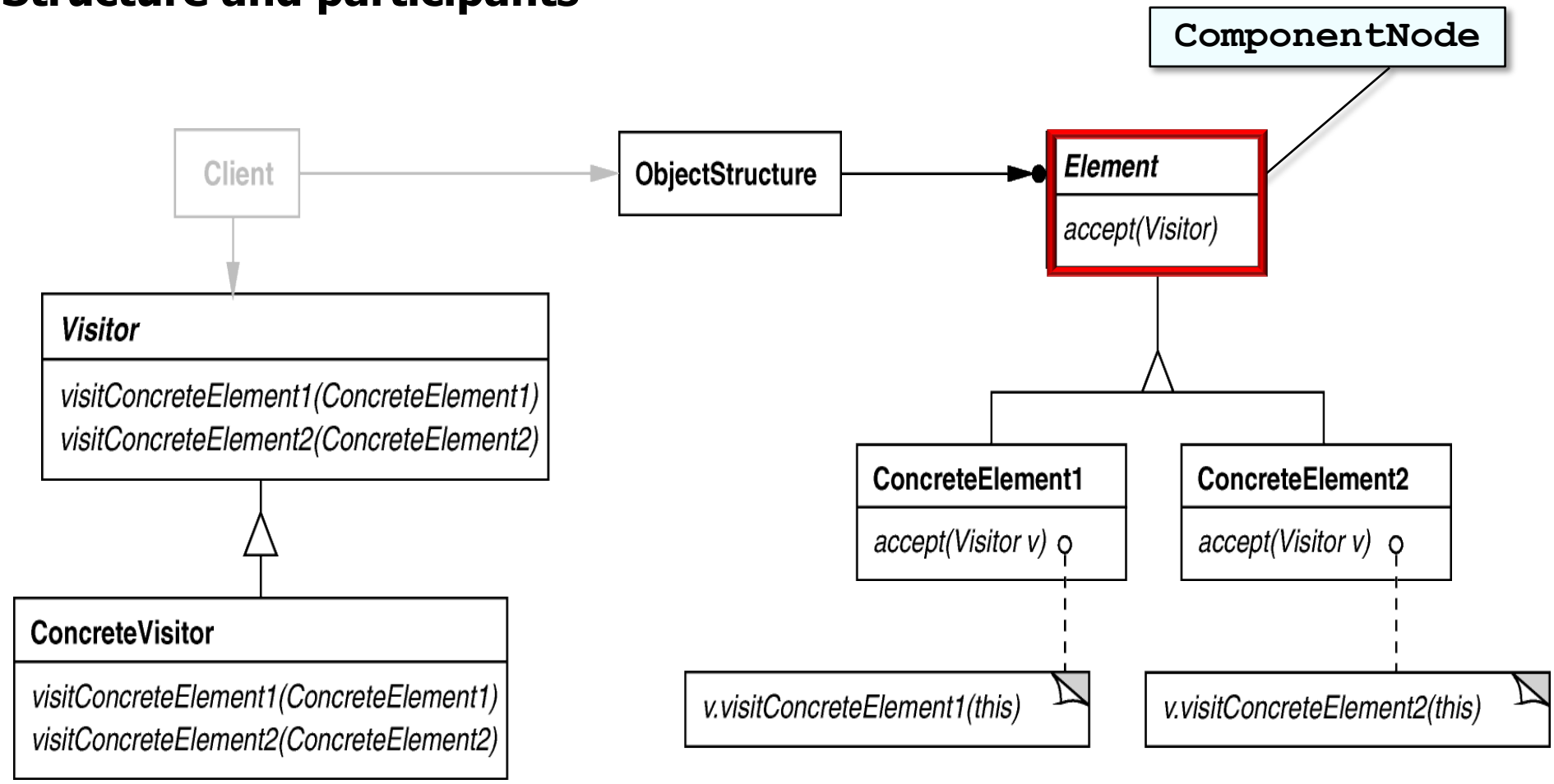
Structure and participants



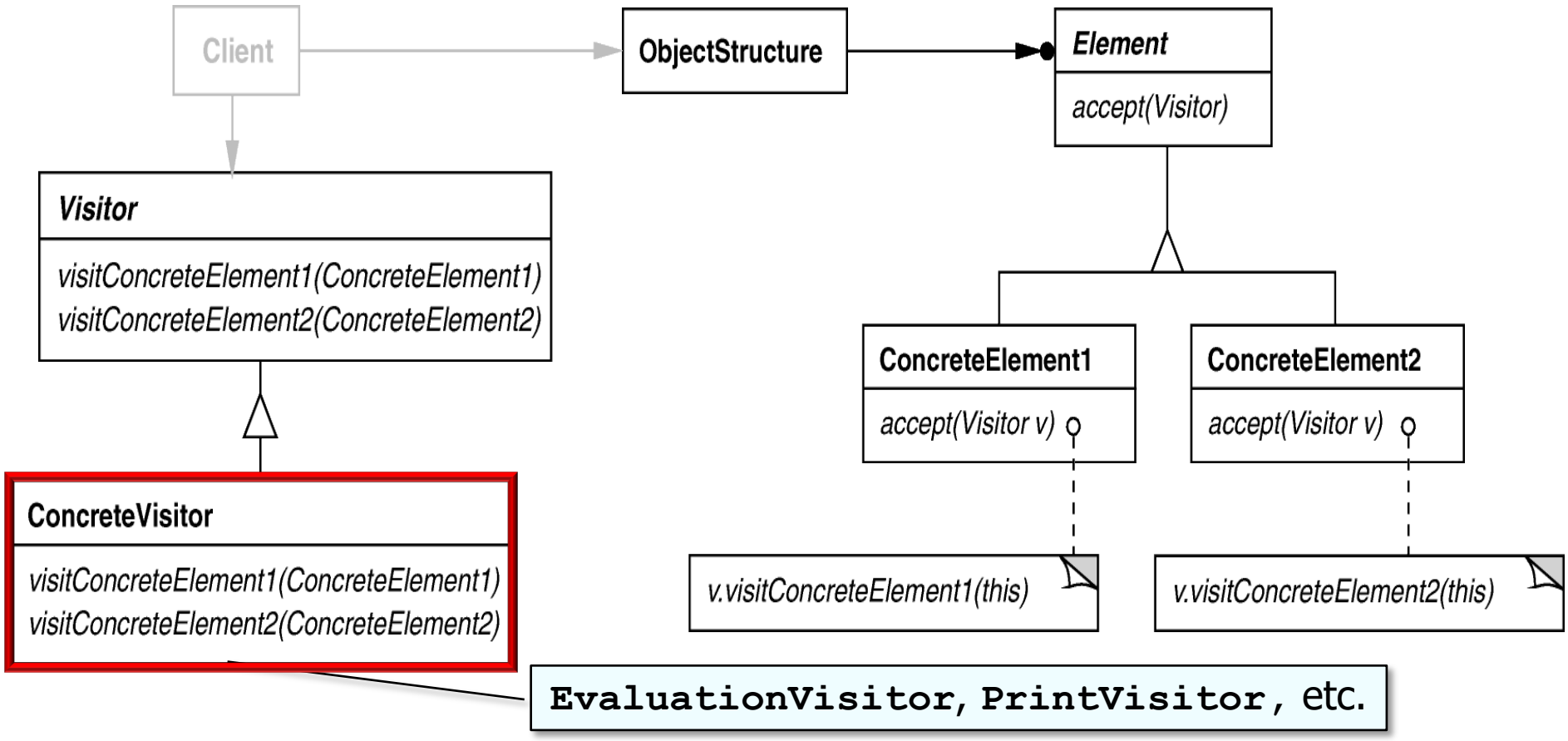
Structure and participants



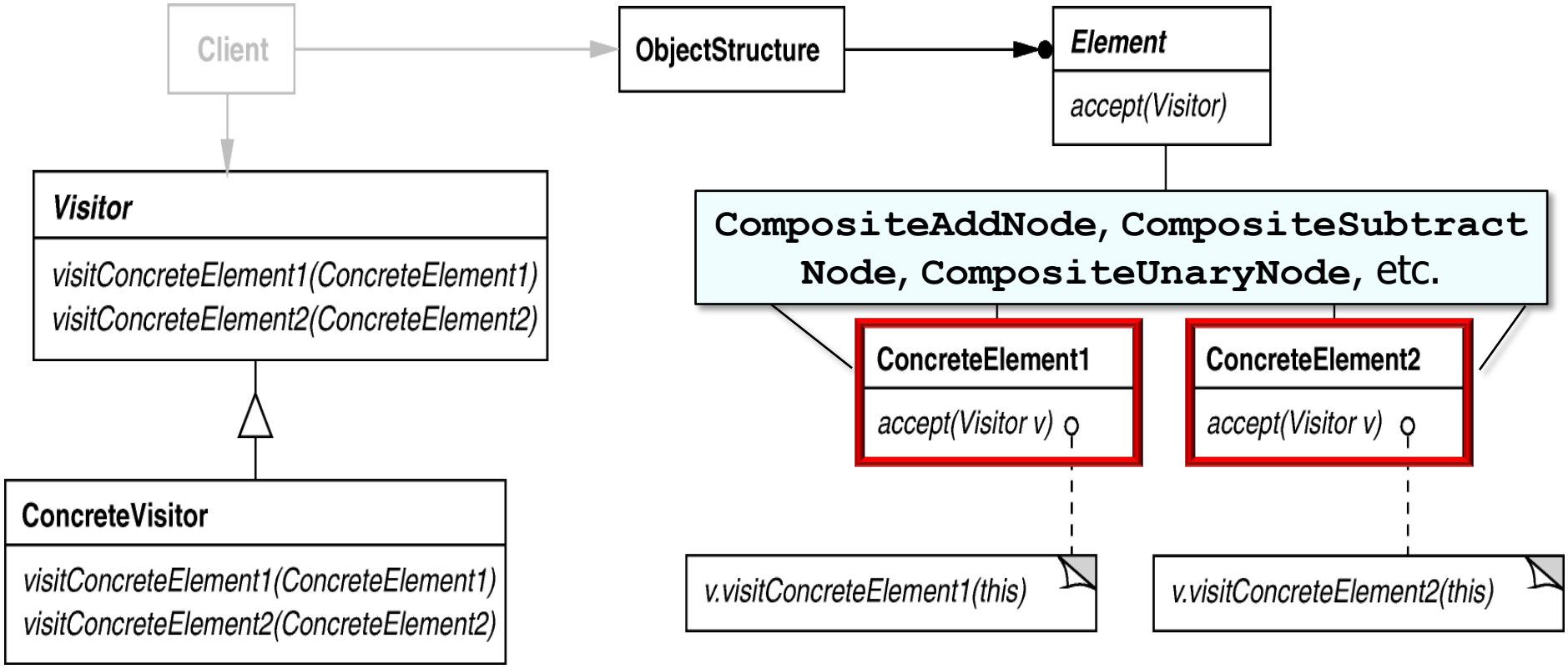
Structure and participants



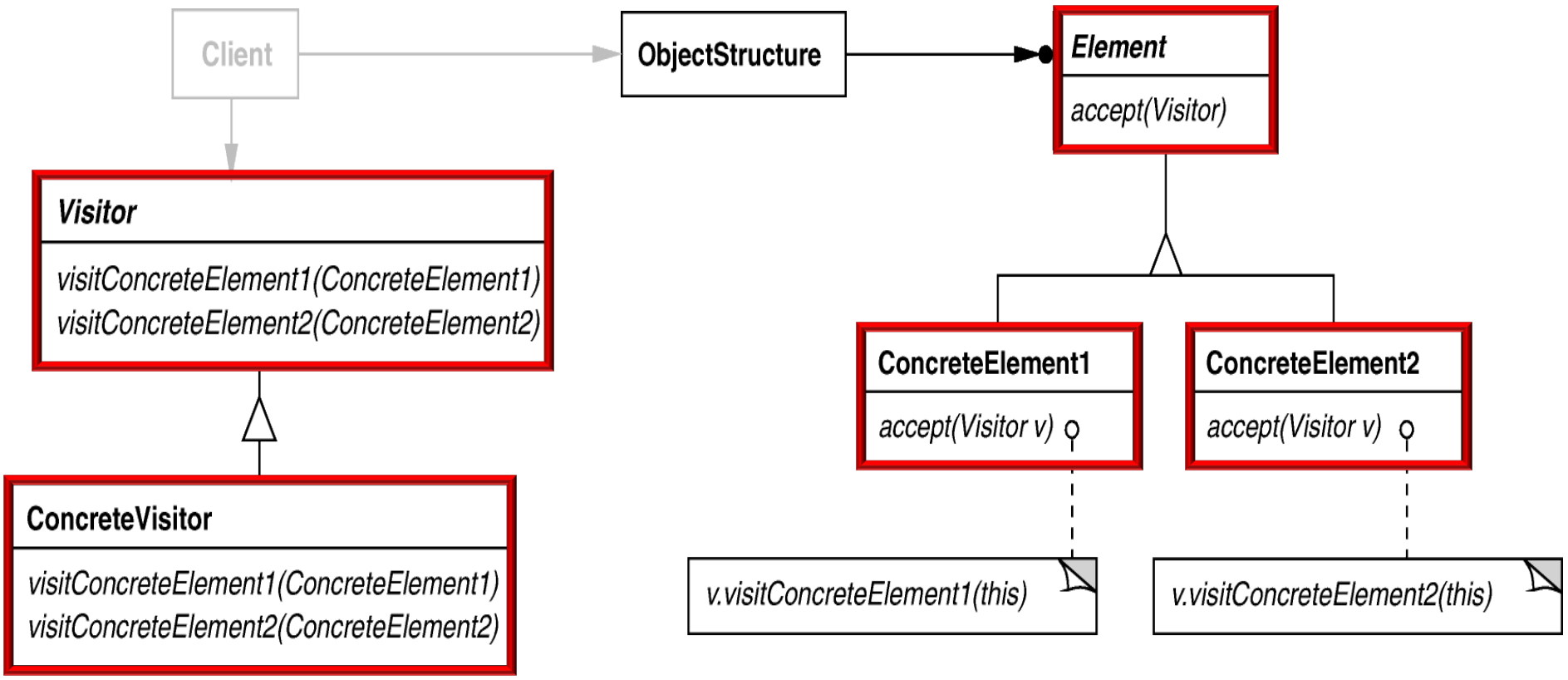
Structure and participants



Structure and participants

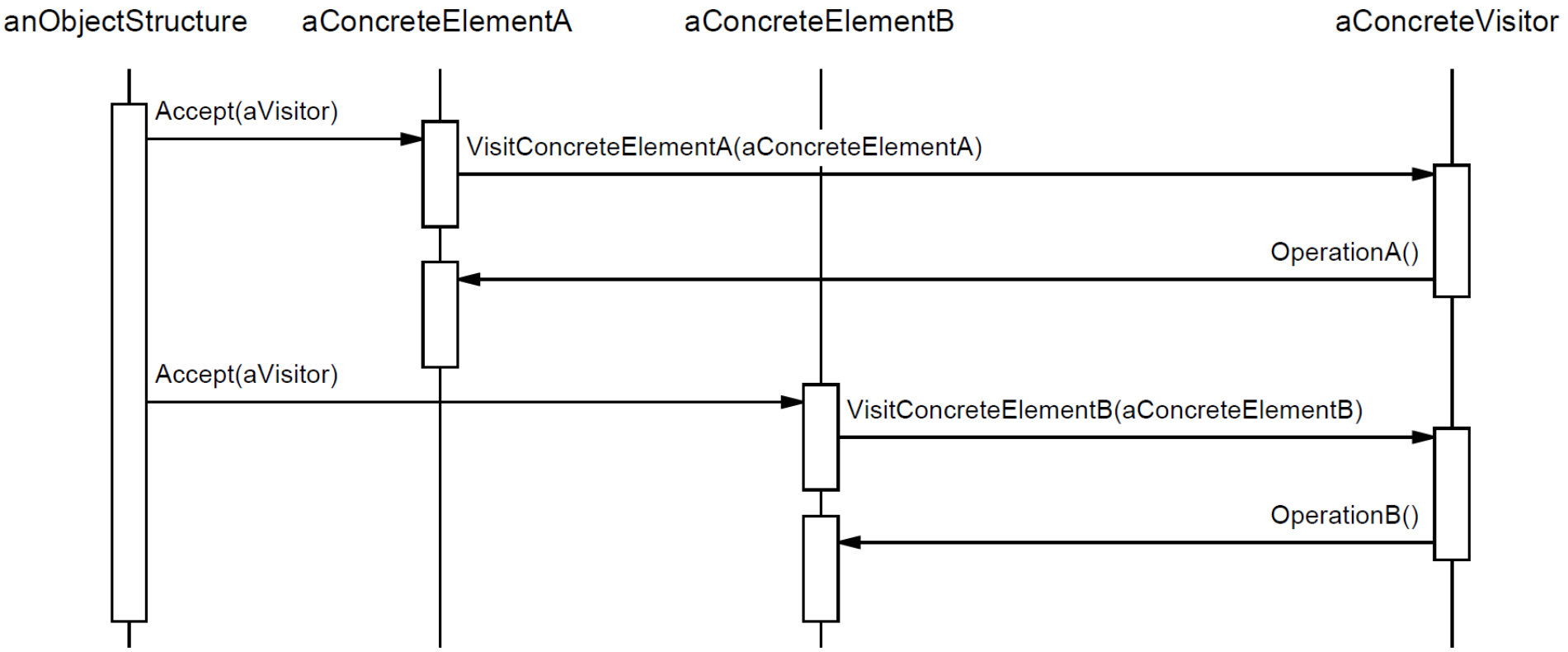


Structure and participants



Visitior's dual inheritance hierarchy + dynamic/static polymorphism is tricky.

Collaborations



This generic object interaction diagram doesn't shed much light on *Visitor*!

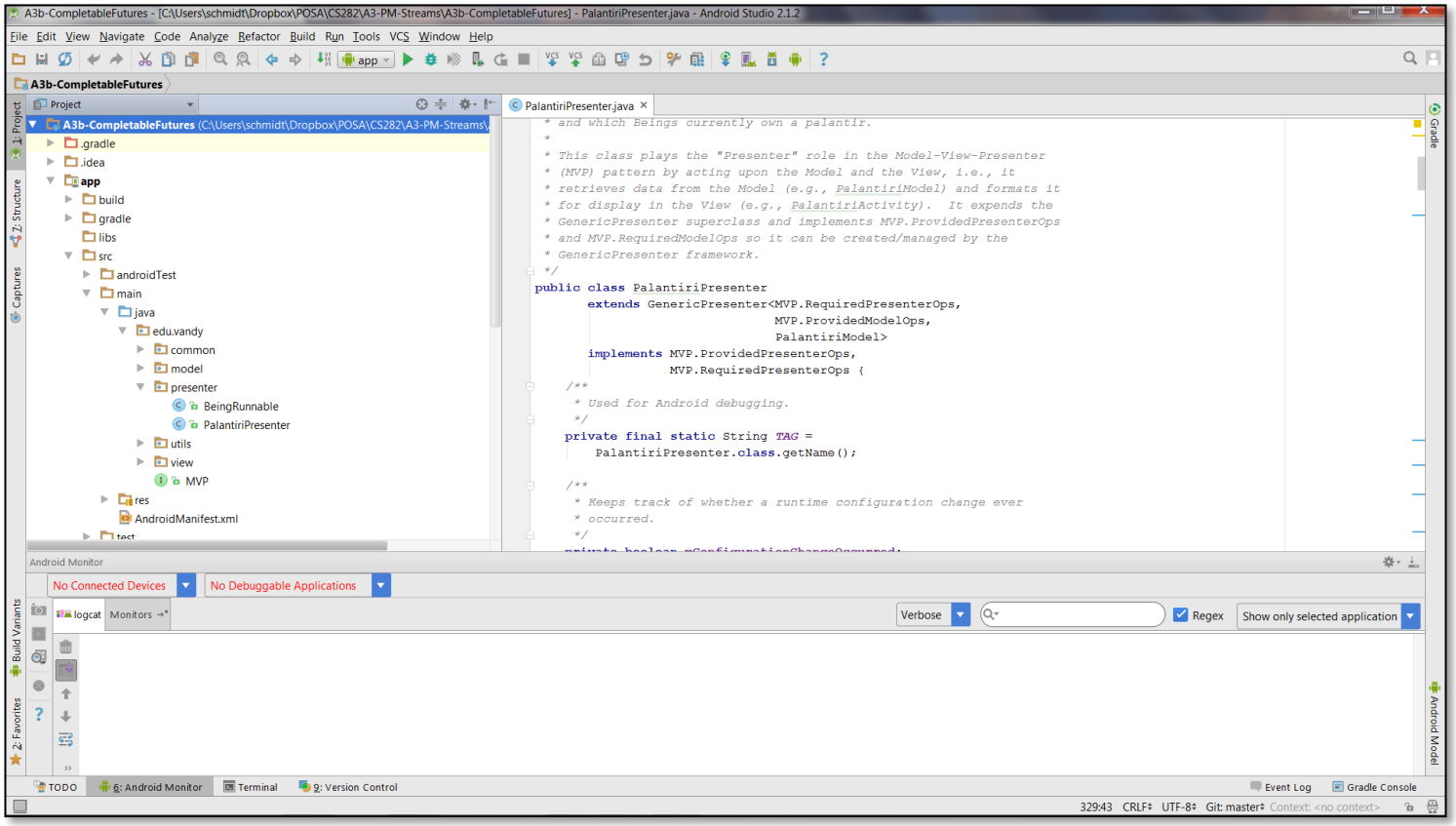
The Visitor Pattern

Implementation in Java

Douglas C. Schmidt

Learning Objectives in This Lesson

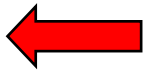
- Recognize how the *Visitor* pattern can be applied to enhance expression tree operation extensibility.
- Understand the *Visitor* pattern.
- Know how to implement the *Visitor* pattern in Java.



Visitor implementation in Java (1/2)

- The **`PrintVisitor`** class prints character code or the value for each node.

```
public class PrintVisitor implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeDivideNode);
  // etc.
};
```



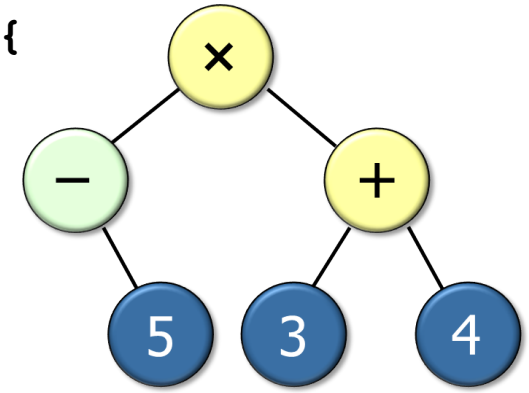
for all relevant `ComponentNode` subclasses

See ExpressionTree/CommandLine/src/expressiontree/visitors

Visitor implementation in Java (1/2)

- The **`PrintVisitor`** class prints character code or the value for each node.

```
public class PrintVisitor implements Visitor {
   public void visit(LeafNode);
   public void visit(CompositeAddNode);
   public void visit(CompositeDivideNode);
   // etc.
};
```



- Can be combined with any traversal ordering algorithm, e.g.,

```
Visitor visitor = visitorFactory.makeVisitor("print");
```

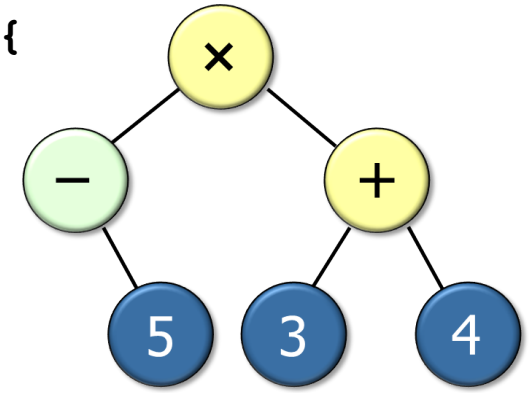
Factory method creates a print visitor

```
ExpressionTree tree = makeExpressionTree("-5 * (3 + 4)");


for(Iterator<ExpressionTree> it = tree.iterator("post-order");
    it.hasNext();)
  it.next().accept(visitor);
```

Visitor implementation in Java (1/2)

- The **`PrintVisitor`** class prints character code or the value for each node.

```
public class PrintVisitor implements Visitor {
   public void visit(LeafNode);
   public void visit(CompositeAddNode);
   public void visit(CompositeDivideNode);
   // etc.
};
```



- Can be combined with any traversal ordering algorithm, e.g.,

```
Visitor visitor = visitorFactory.makeVisitor("print");


ExpressionTree tree = makeExpressionTree("-5 * (3 + 4)");
```

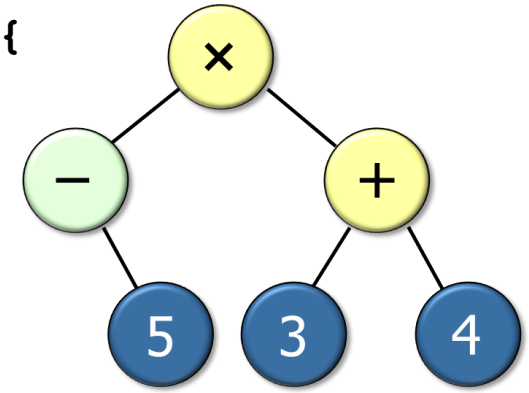
Factory method creates an expression tree

```
for(Iterator<ExpressionTree> it = tree.iterator("post-order");
    it.hasNext();)
  it.next().accept(visitor);
```

Visitor implementation in Java (1/2)

- The **`PrintVisitor`** class prints character code or the value for each node.

```
public class PrintVisitor implements Visitor {
   public void visit(LeafNode);
   public void visit(CompositeAddNode);
   public void visit(CompositeDivideNode);
   // etc.
};
```



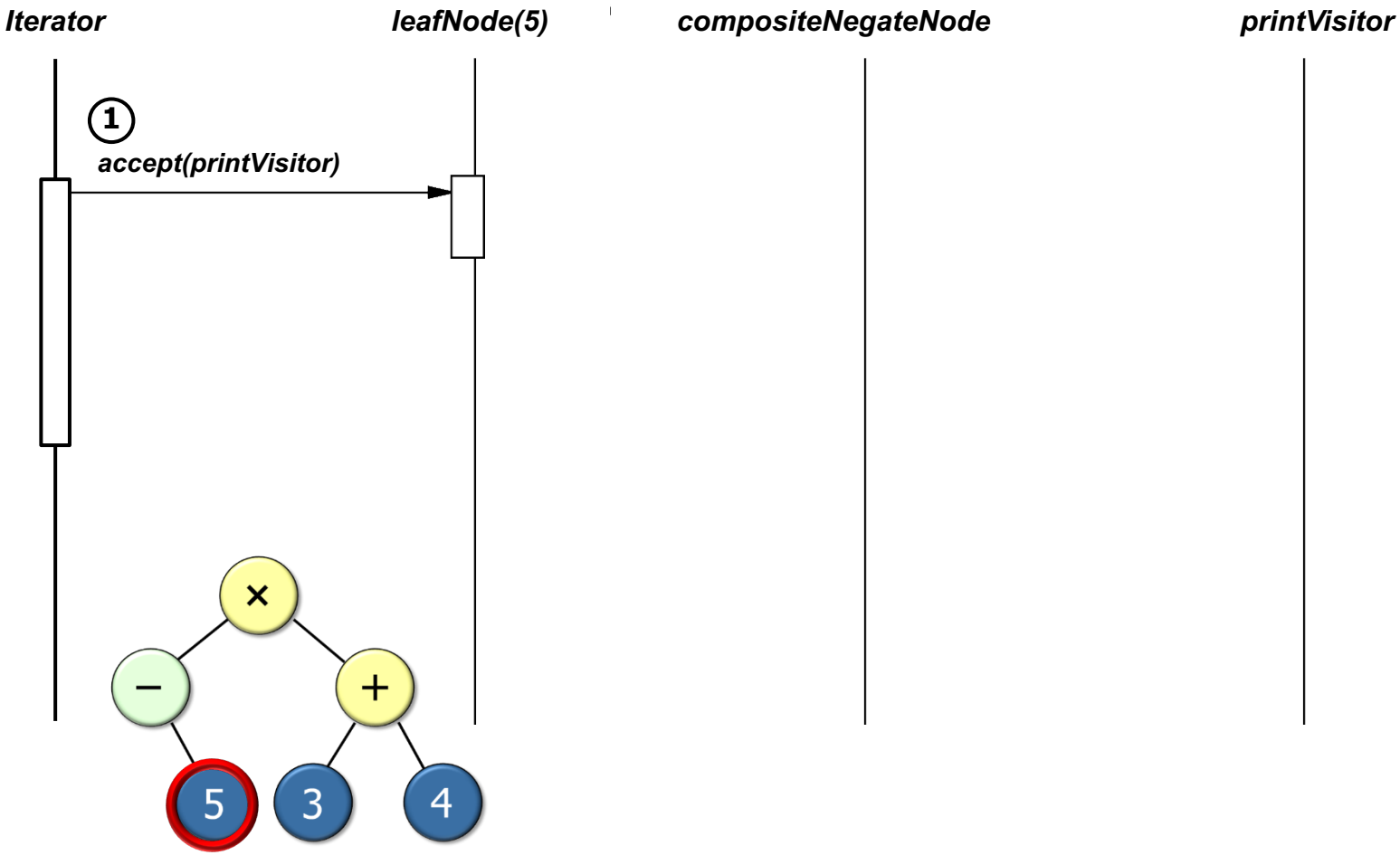
- Can be combined with any traversal ordering algorithm, e.g.,

```
Visitor visitor = visitorFactory.makeVisitor("print");


ExpressionTree tree = makeExpressionTree("-5 * (3 + 4)");


for(Iterator<ExpressionTree> it = tree.iterator("post-order");
    it.hasNext();)
  it.next().accept(visitor);
```

`accept()` forwards to implementor's `accept()`, which calls `visit(this)`

Visitor implementation in Java (1/2)

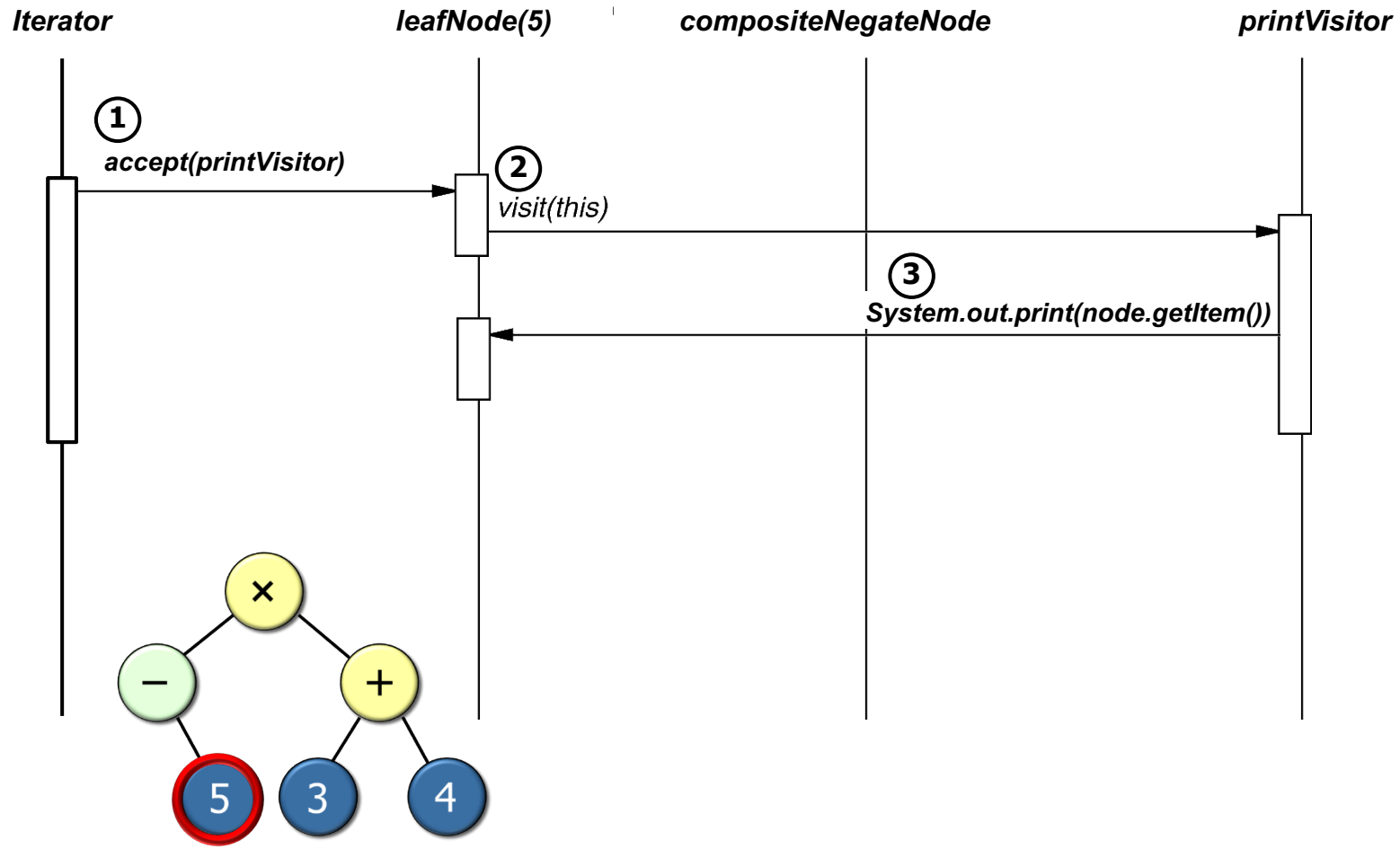
- Iterator controls the order in which `accept()` is called on each node in the tree.



This example is based on a "post-order" traversal.

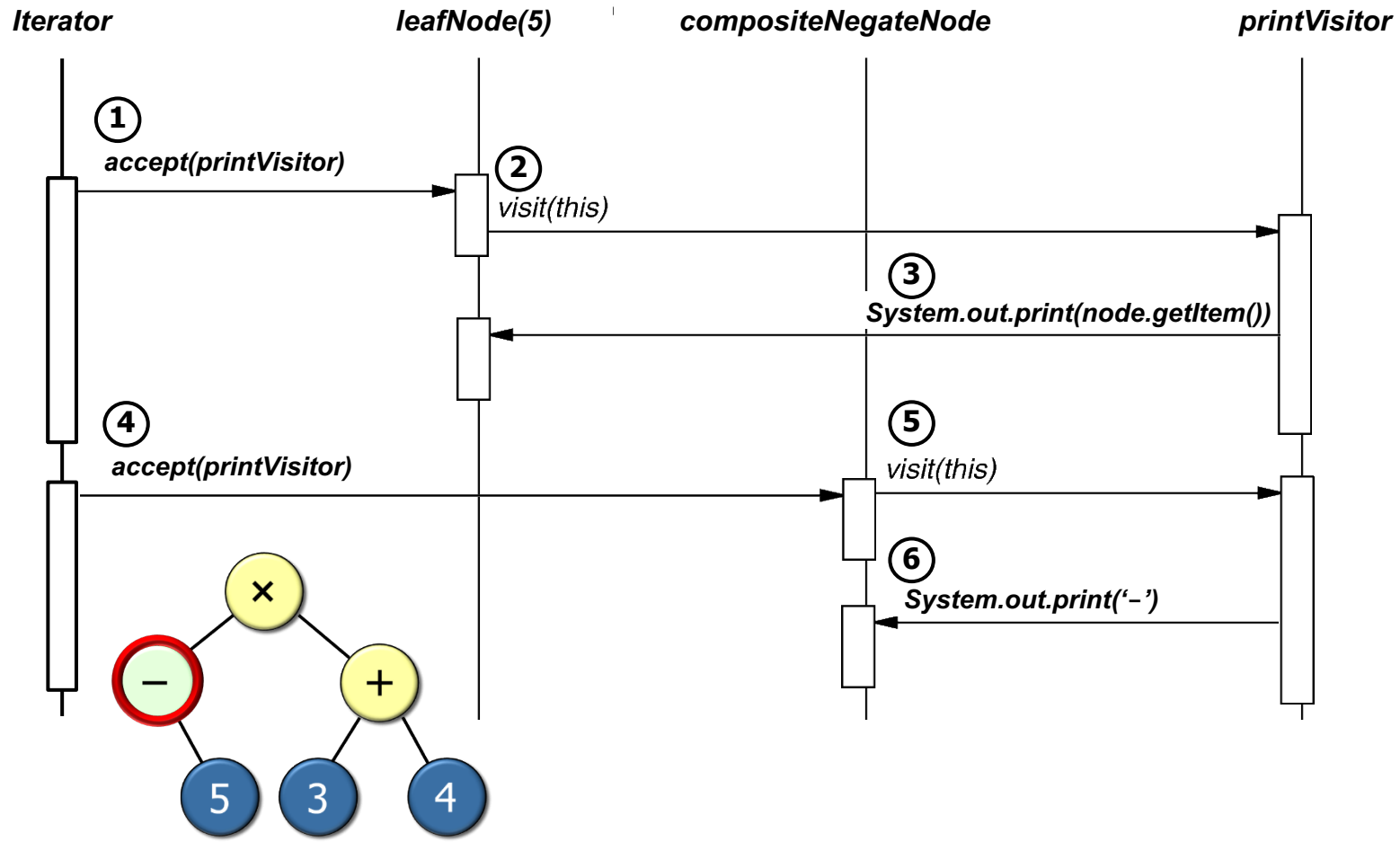
Visitor implementation in Java (1/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired print action.



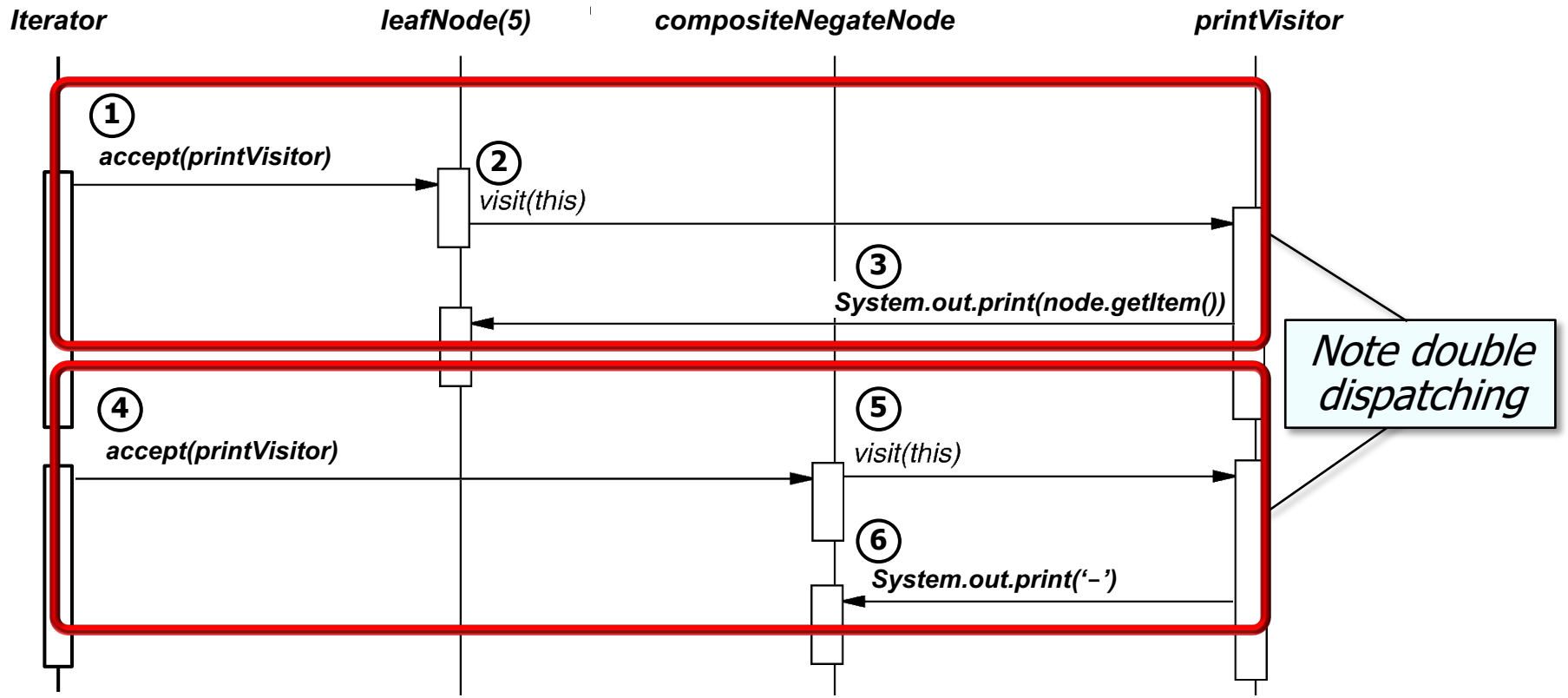
Visitor implementation in Java (1/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired print action.



Visitor implementation in Java (1/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired print action.

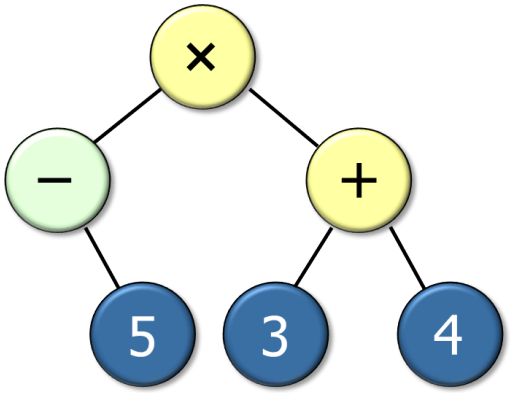


en.wikipedia.org/wiki/Double_dispatch has more info on double dispatching.

Visitor implementation in Java (2/2)

- **`EvaluationVisitor`** visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
        implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
```

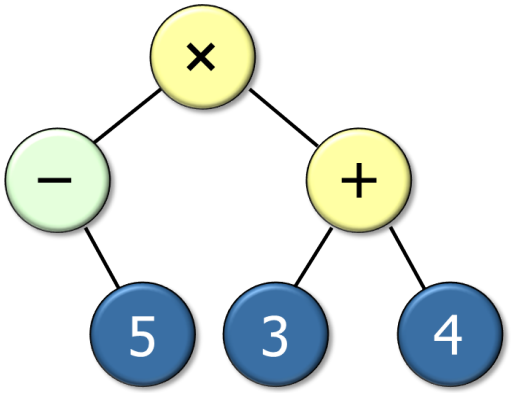


5~34+×

Visitor implementation in Java (2/2)

- **`EvaluationVisitor`** visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
       implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
  private Stack<Integer>
     mStack = new Stack<>();
}
```



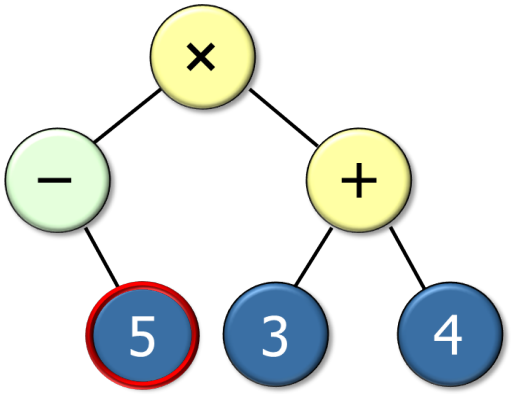
5~34+×

This stack stores the post-order expression tree values processed incrementally during the iteration traversal.

Visitor implementation in Java (2/2)

- **`EvaluationVisitor`** visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
       implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
  private Stack<Integer>
     mStack = new Stack<>();
}
```



5~34+×

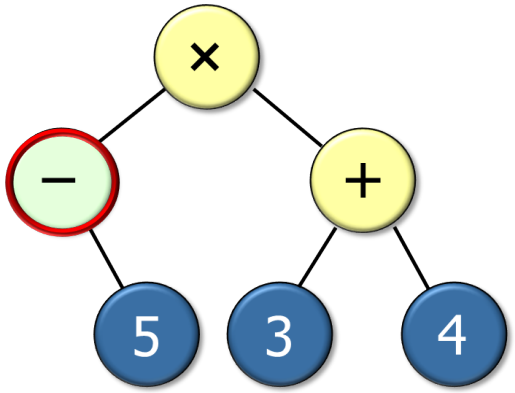
visit() method behavior

1. S = [5] `push(node.getItem())`

Visitor implementation in Java (2/2)

- **`EvaluationVisitor`** visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
        implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
  private Stack<Integer>
     mStack = new Stack<>();
}
```



`5~34+×`

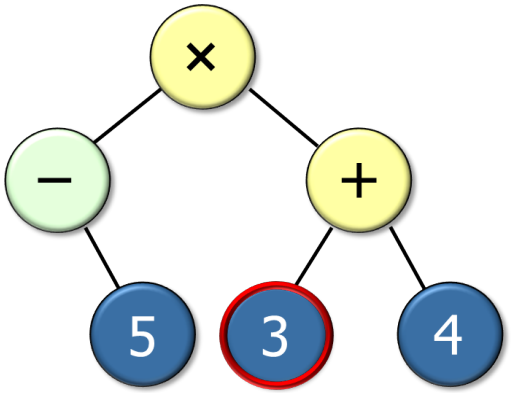
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`

Visitor implementation in Java (2/2)

- **`EvaluationVisitor`** visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
        implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
  private Stack<Integer>
     mStack = new Stack<>();
}
```



5~34+×

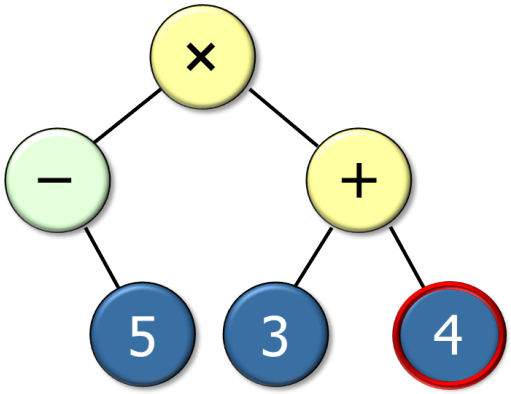
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`
3. S = [-5, 3] **`push(node.getItem())`**

Visitor implementation in Java (2/2)

- **`EvaluationVisitor`** visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
        implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
  private Stack<Integer>
     mStack = new Stack<>();
}
```



5~34+×

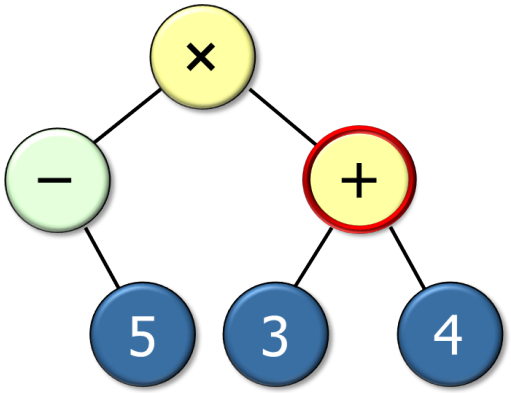
visit() method behavior

1. S = [5] `push(node.getItem())`
2. S = [-5] `push(-pop())`
3. S = [-5, 3] `push(node.getItem())`
4. S = [-5, 3, 4] **`push(node.getItem())`**

Visitor implementation in Java (2/2)

- **`EvaluationVisitor`** visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
        implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
  private Stack<Integer>
     mStack = new Stack<>();
}
```



5~34+×

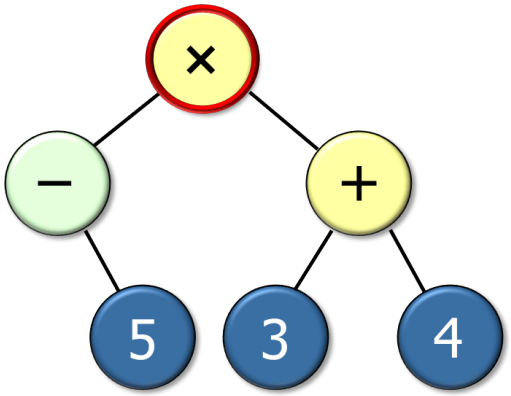
visit() method behavior

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

Visitor implementation in Java (2/2)

- **`EvaluationVisitor`** visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
        implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
  private Stack<Integer>
     mStack = new Stack<>();
}
```



5~34+×

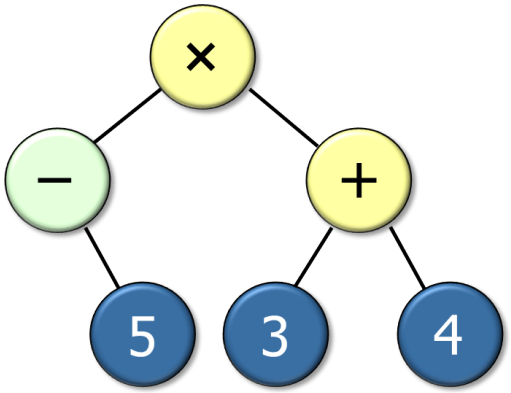
visit() method behavior

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

Visitor implementation in Java (2/2)

- `EvaluationVisitor` visits nodes via a *post-order* iterator to compute yield

```
public class EvaluationVisitor
       implements Visitor {
  public void visit(LeafNode);
  public void visit(CompositeNegateNode);
  public void visit(CompositeAddNode);
  public void visit(CompositeMultipleNode)
  // etc.
  ...
  private Stack<Integer>
     mStack = new Stack<>();
}
```



5~34+×

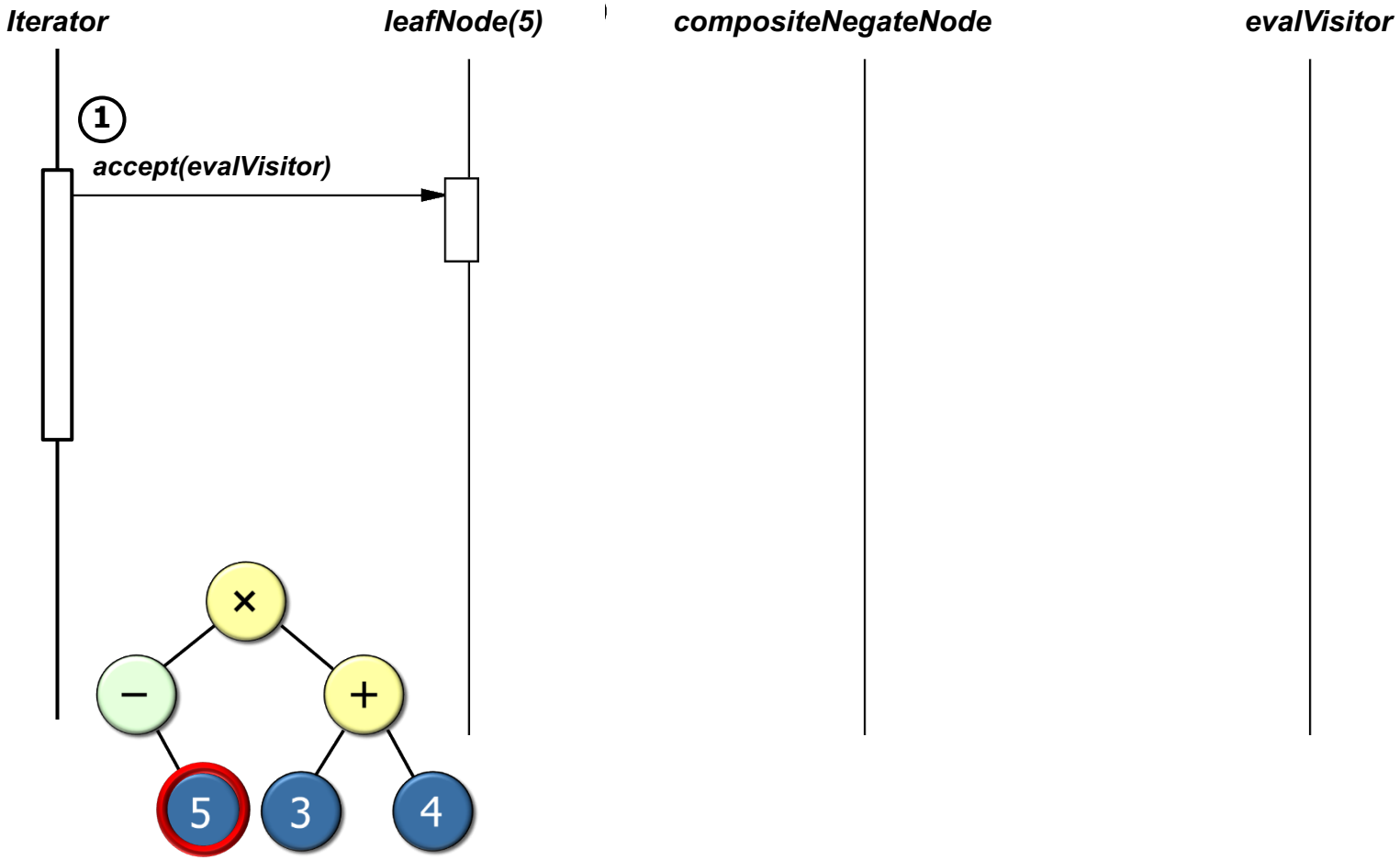
visit() method behavior

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

The final top stack item contains the "yield" of the expression tree.

Visitor implementation in Java (2/2)

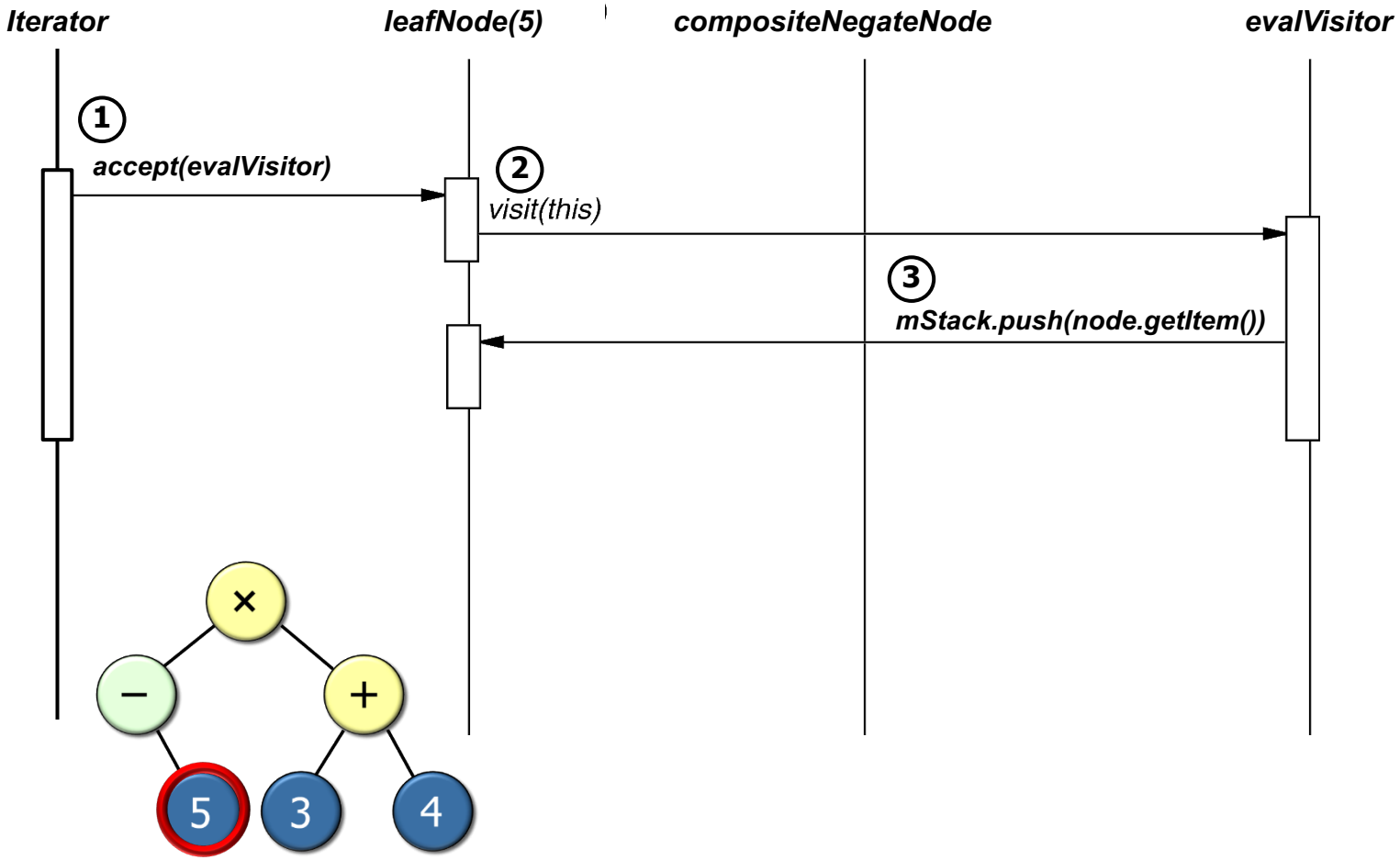
- Iterator controls the order in which `accept()` is called on each node in the tree.



This example is based on a "post-order" traversal.

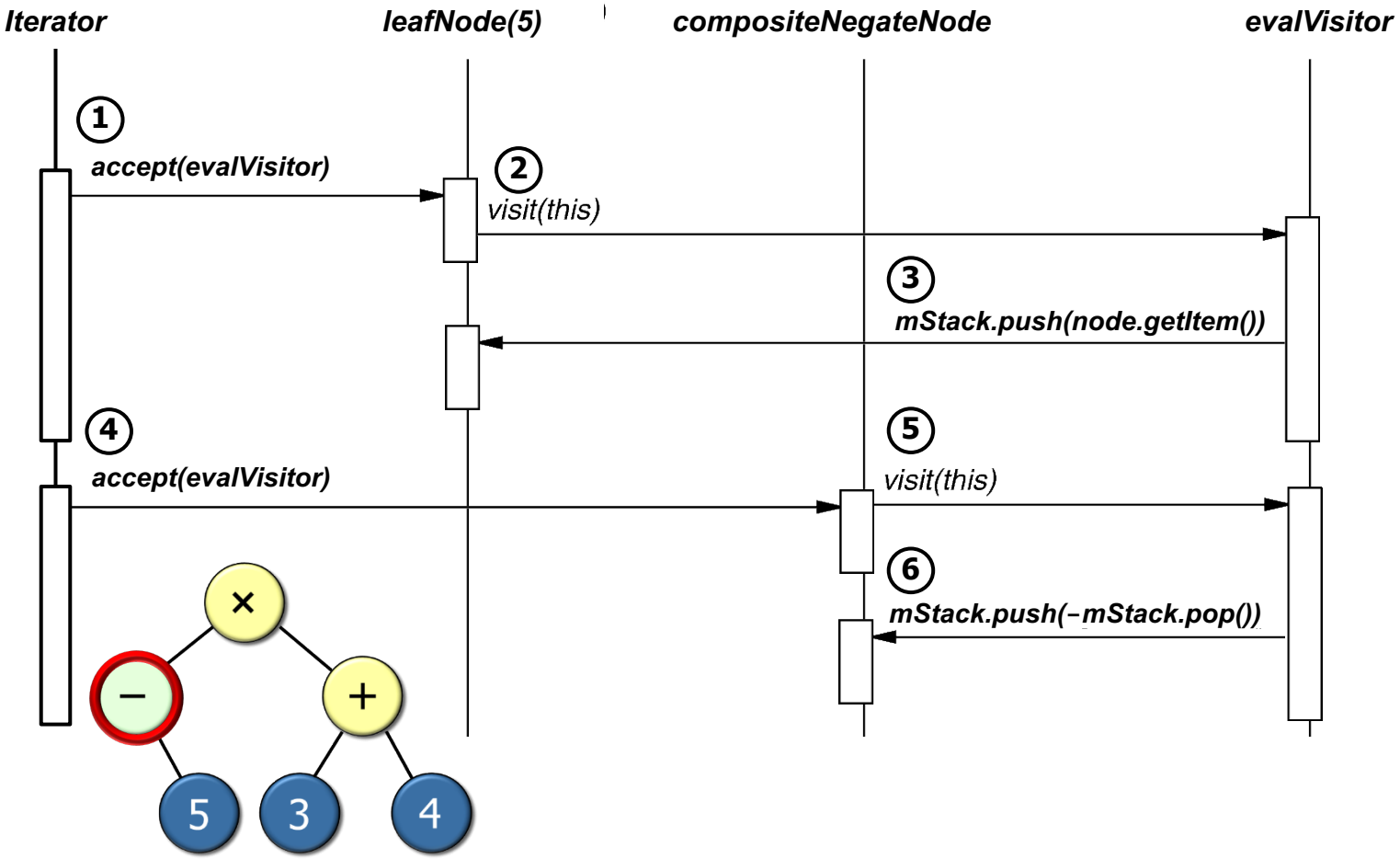
Visitor implementation in Java (2/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired evaluation action.



Visitor implementation in Java (2/2)

- Iterator controls the order in which `accept()` is called on each node in the tree.
- `accept()` then "visits" the node to perform the desired evaluation action.



The Visitor Pattern

Other Considerations

Douglas C. Schmidt

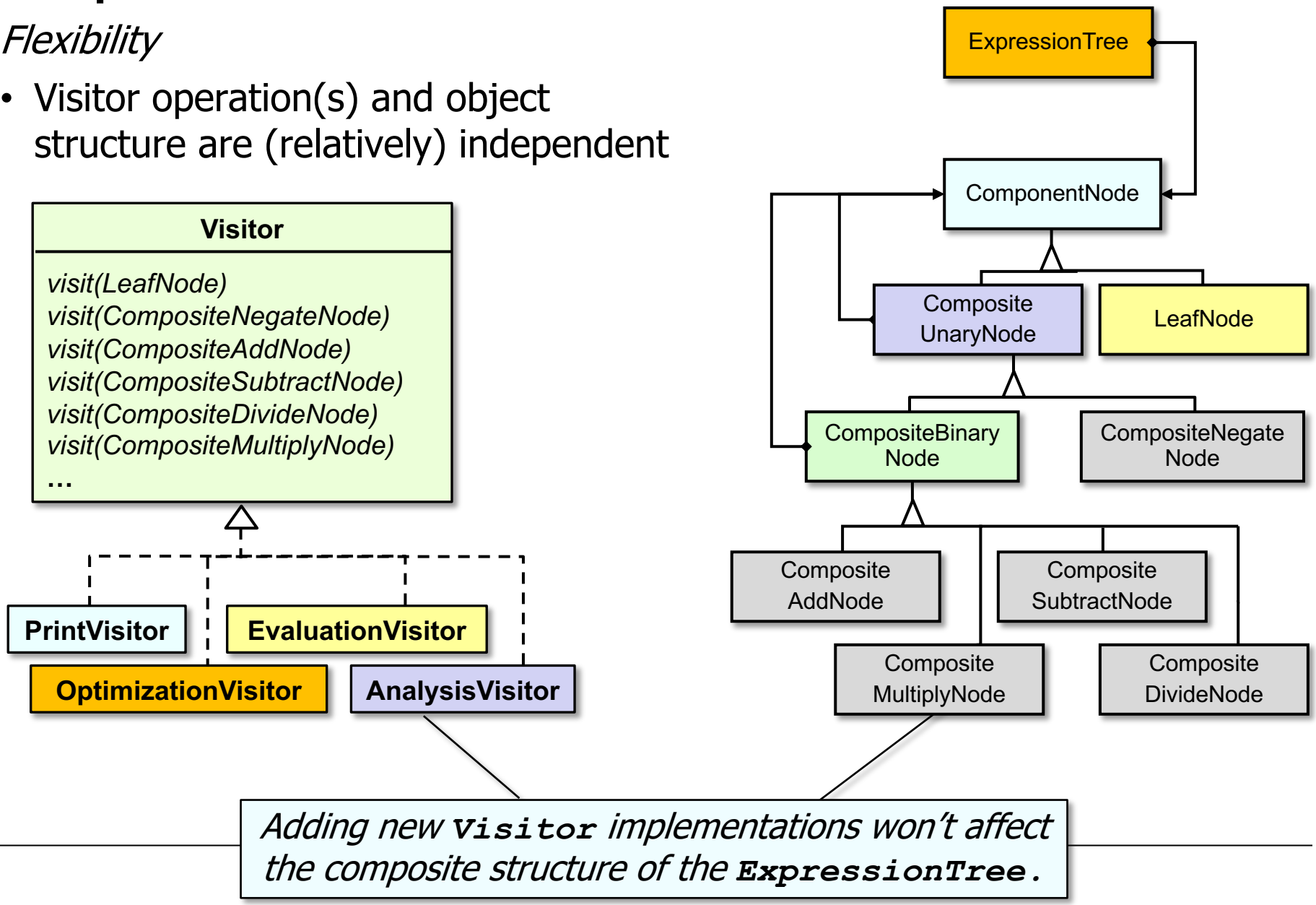
Learning Objectives in This Lesson

- Recognize how the *Visitor* pattern can be applied to enhance expression tree operation extensibility.
- Understand the *Visitor* pattern.
- Know how to implement the *Visitor* pattern in Java.
- Be aware of other considerations when applying the *Visitor* pattern.

Consequences

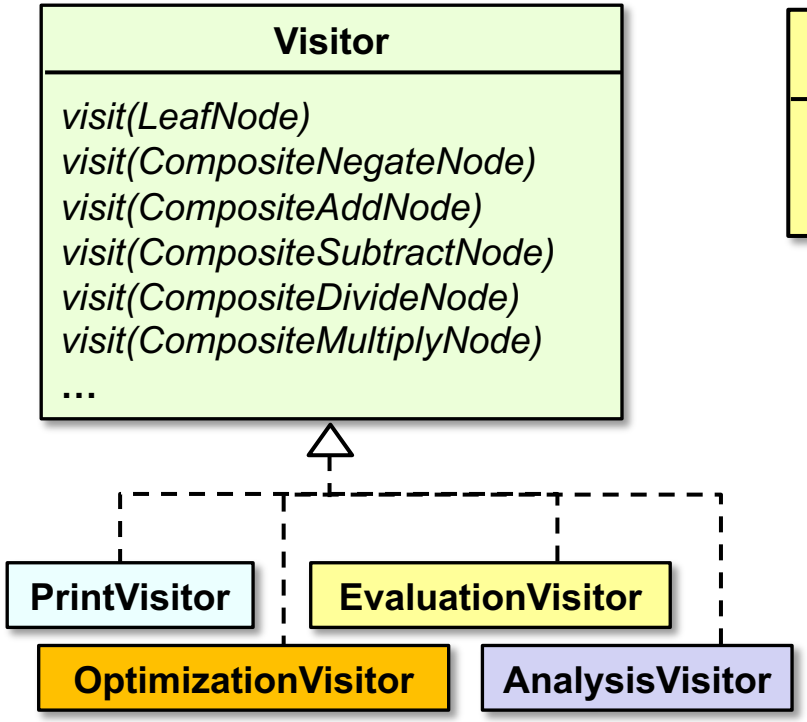
+ *Flexibility*

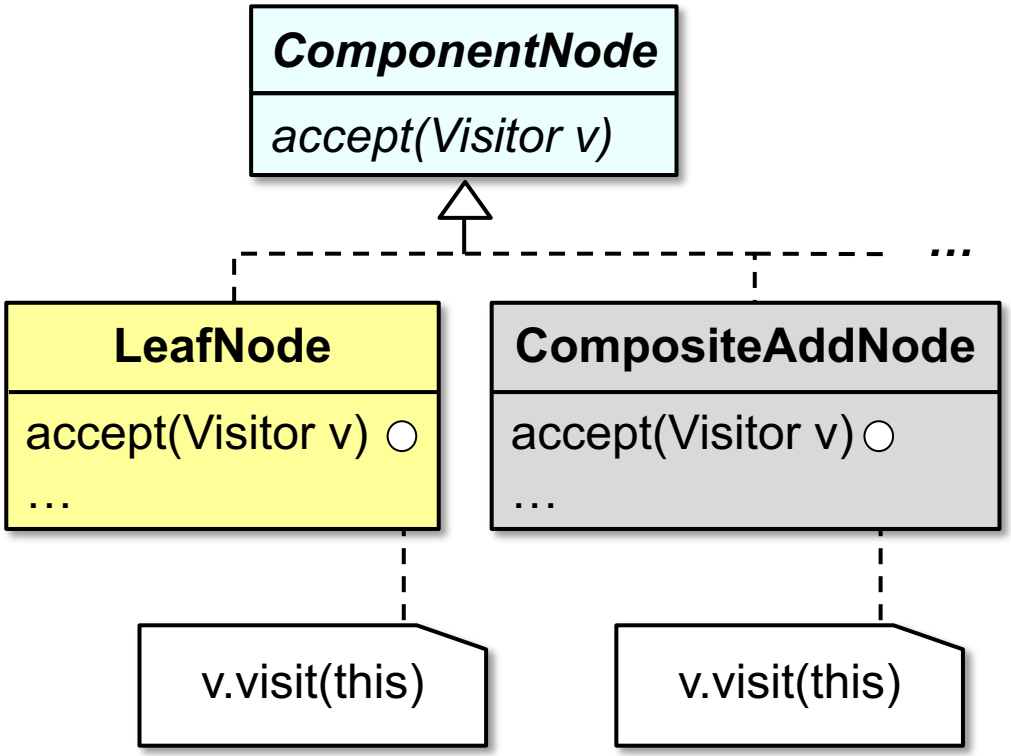
- Visitor operation(s) and object structure are (relatively) independent



Consequences

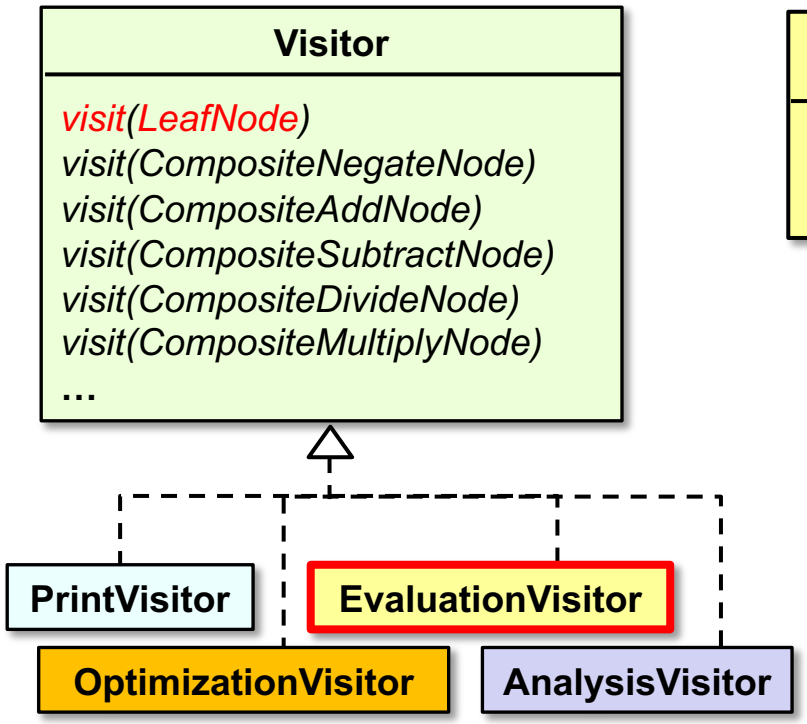
+ *Separation of concerns*
  - Localized functionality in the visitor implementation

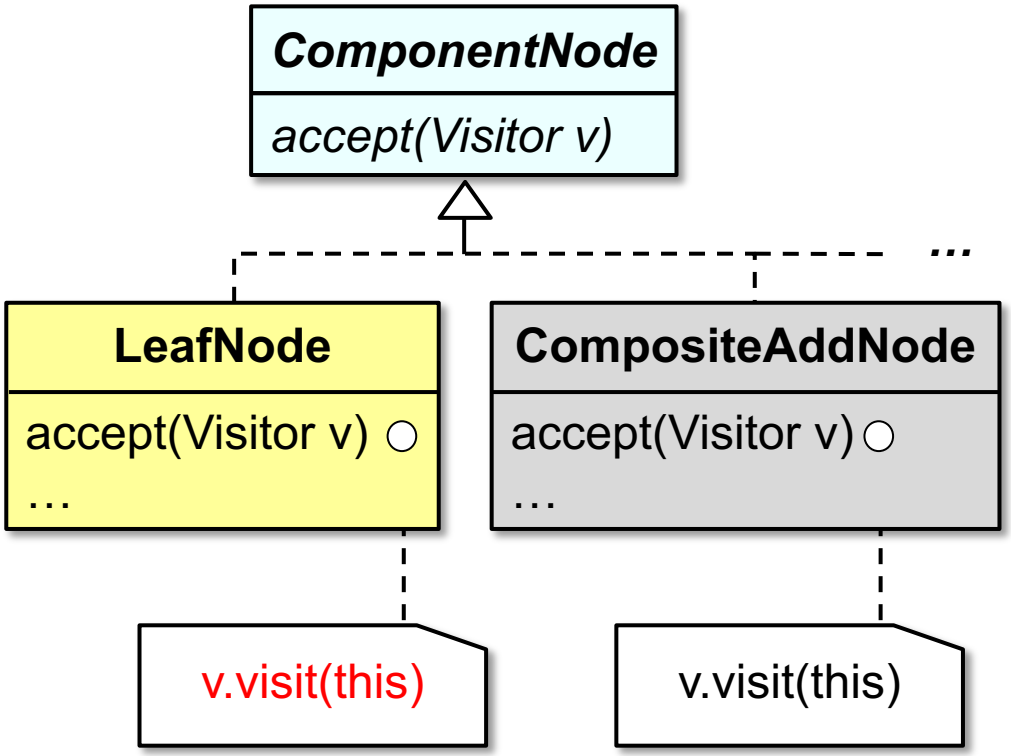




Consequences

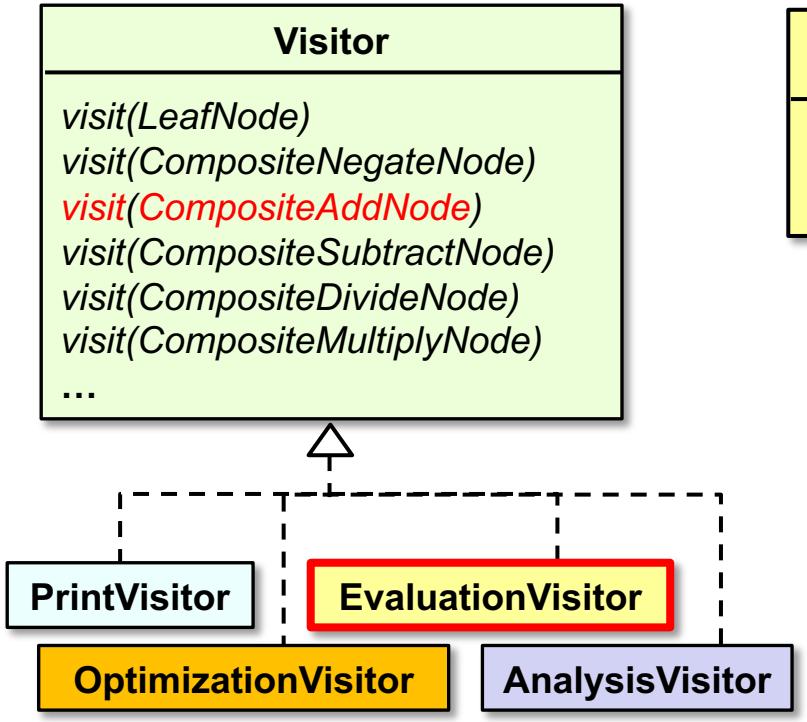
+ *Separation of concerns*
  - Localized functionality in the visitor implementation

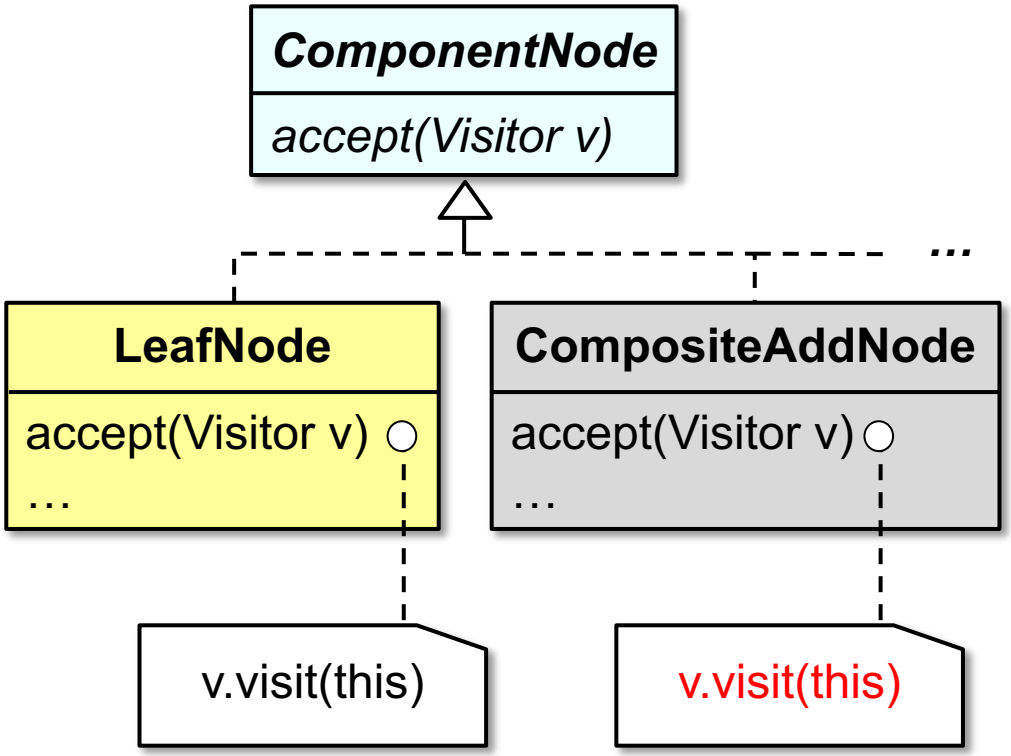




Consequences

+ *Separation of concerns*
  - Localized functionality in the visitor implementation

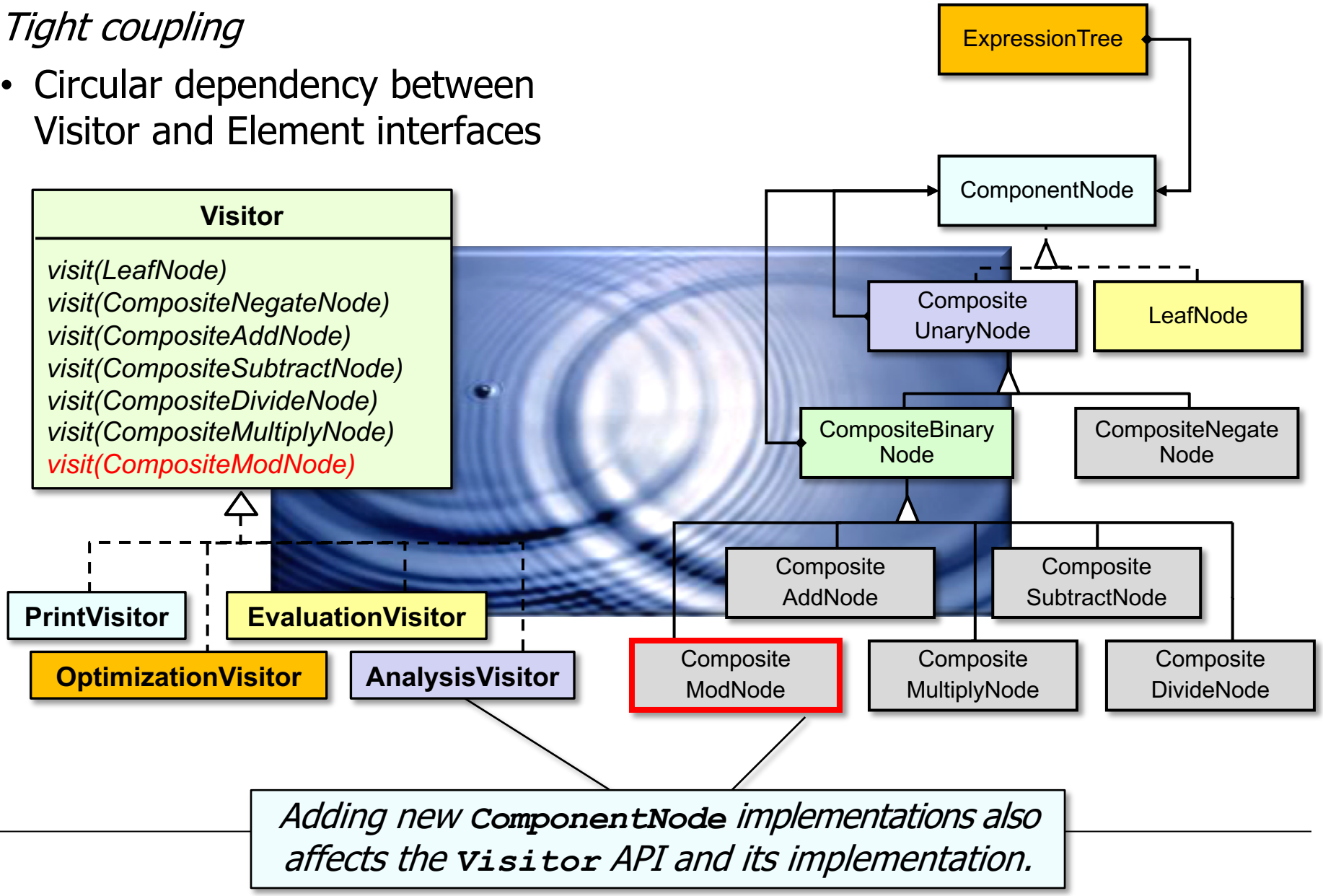




Consequences

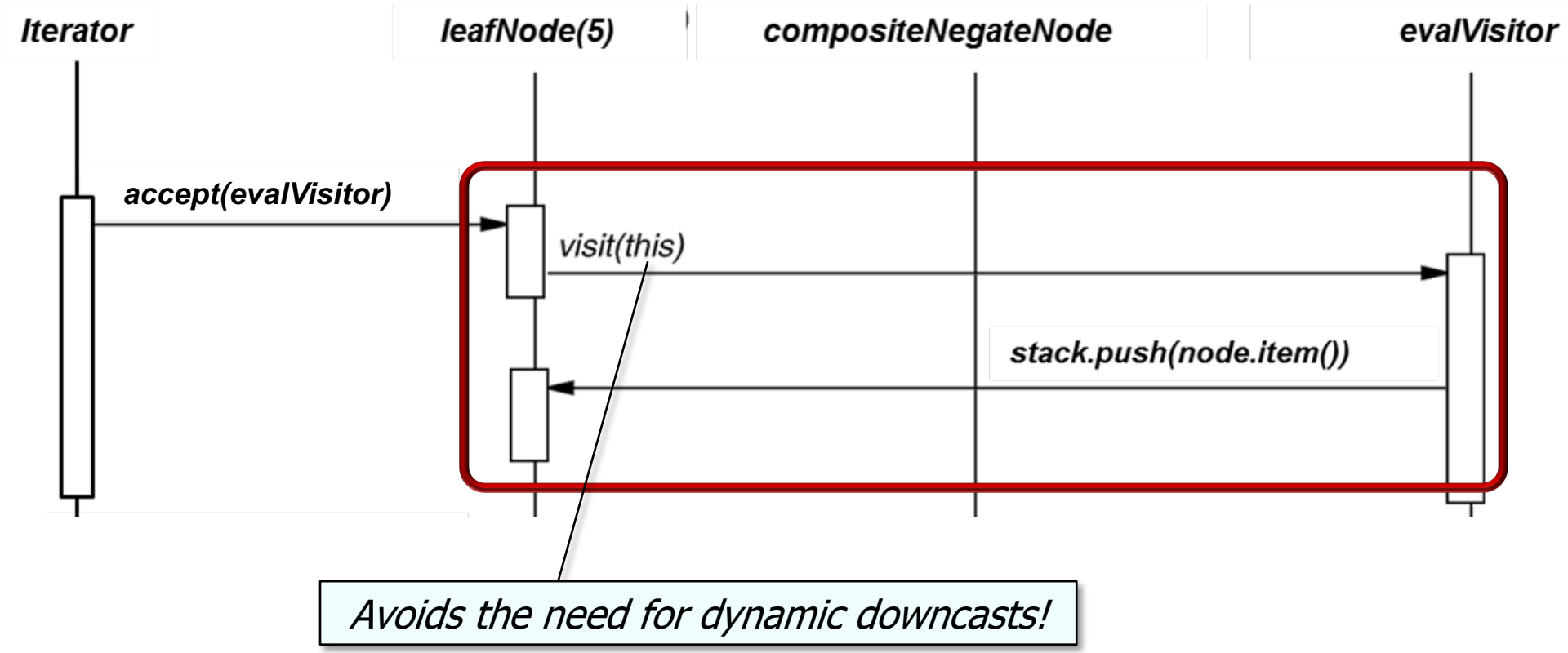
– *Tight coupling*

- Circular dependency between Visitor and Element interfaces



Implementation considerations

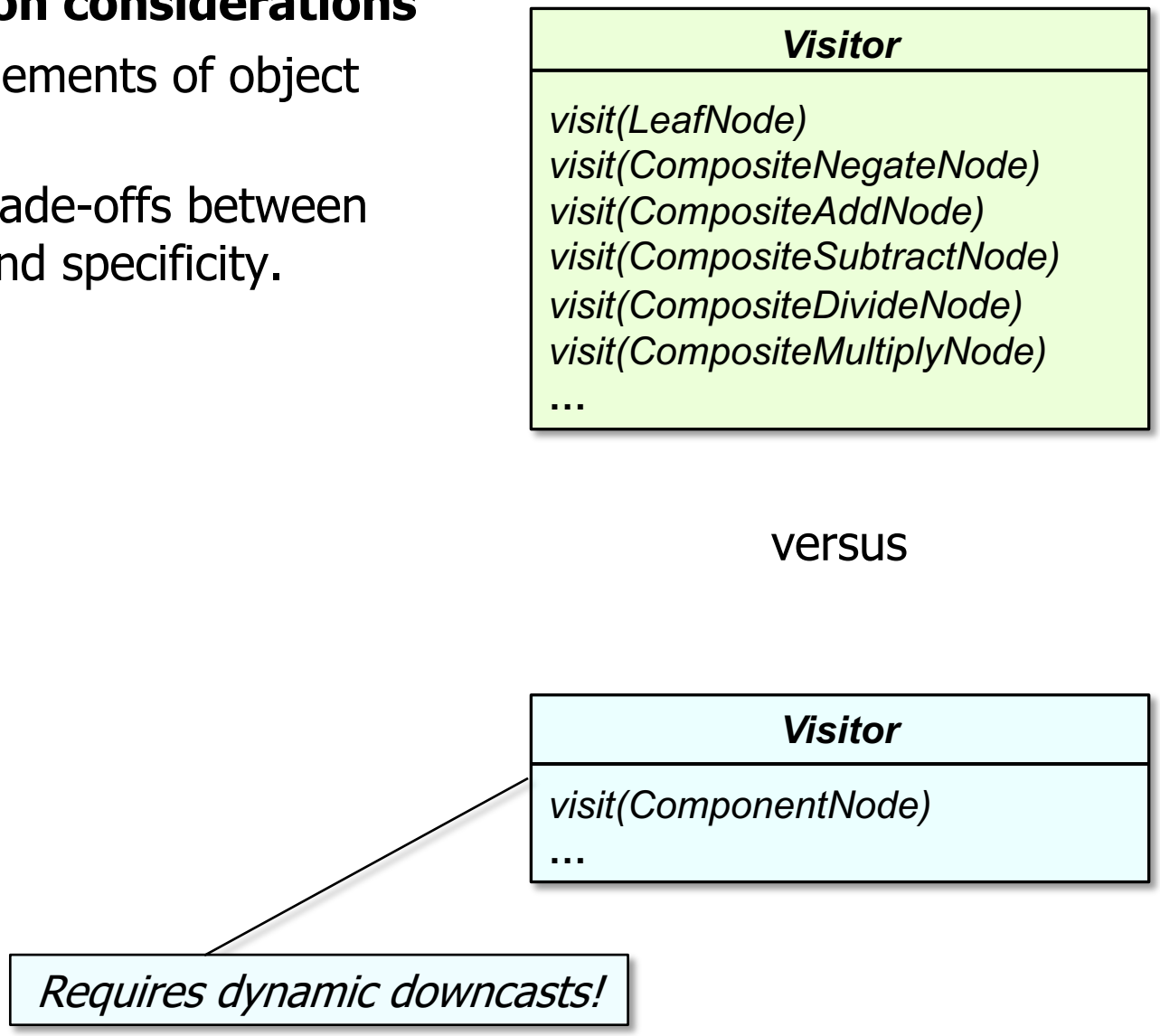
- *Double dispatch*
  - A mechanism that dispatches a method call to different concrete methods depending on runtime types of two objects involved in the call



en.wikipedia.org/wiki/Double_dispatch has more information on *Double Dispatch.*

Implementation considerations

- Interface to elements of object structure
  - There are trade-offs between generality and specificity.



Known uses

- ProgramNodeEnumerator in Smalltalk-80 compiler
- IRIS Inventor scene rendering
- java.nio.file.FileVisitor and SimpleFileVisitor

Interface FileVisitor<T>

All Known Implementing Classes:
SimpleFileVisitor

---

public interface **FileVisitor<T>**

A visitor of files. An implementation of this interface is provided to the Files.walkFileTree methods to visit each file in a file tree.

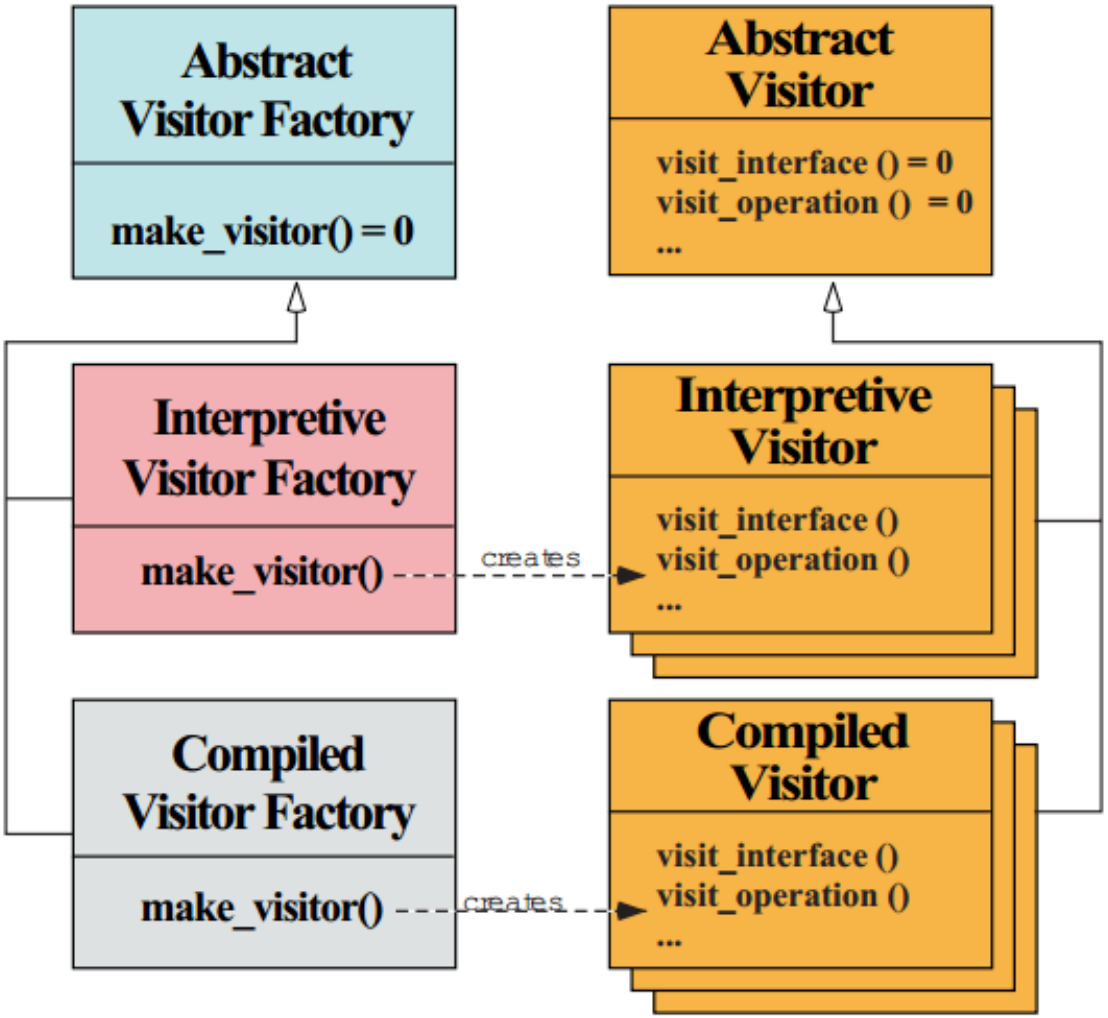
Usage Examples: Suppose we want to delete a file tree. In that case, each directory should be deleted after the entries in the directory are deleted.

```
Path start = ...
Files.walkFileTree(start, new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException
    {
        Files.delete(file);
        return FileVisitResult.CONTINUE;
    }
    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException e)
        throws IOException
    {
        if (e == null) {
            Files.delete(dir);
            return FileVisitResult.CONTINUE;
        } else {
            // directory iteration failed
            throw e;
```

See docs.oracle.com/javase/8/docs/api/java/nio/file/FileVisitor.html

Known uses

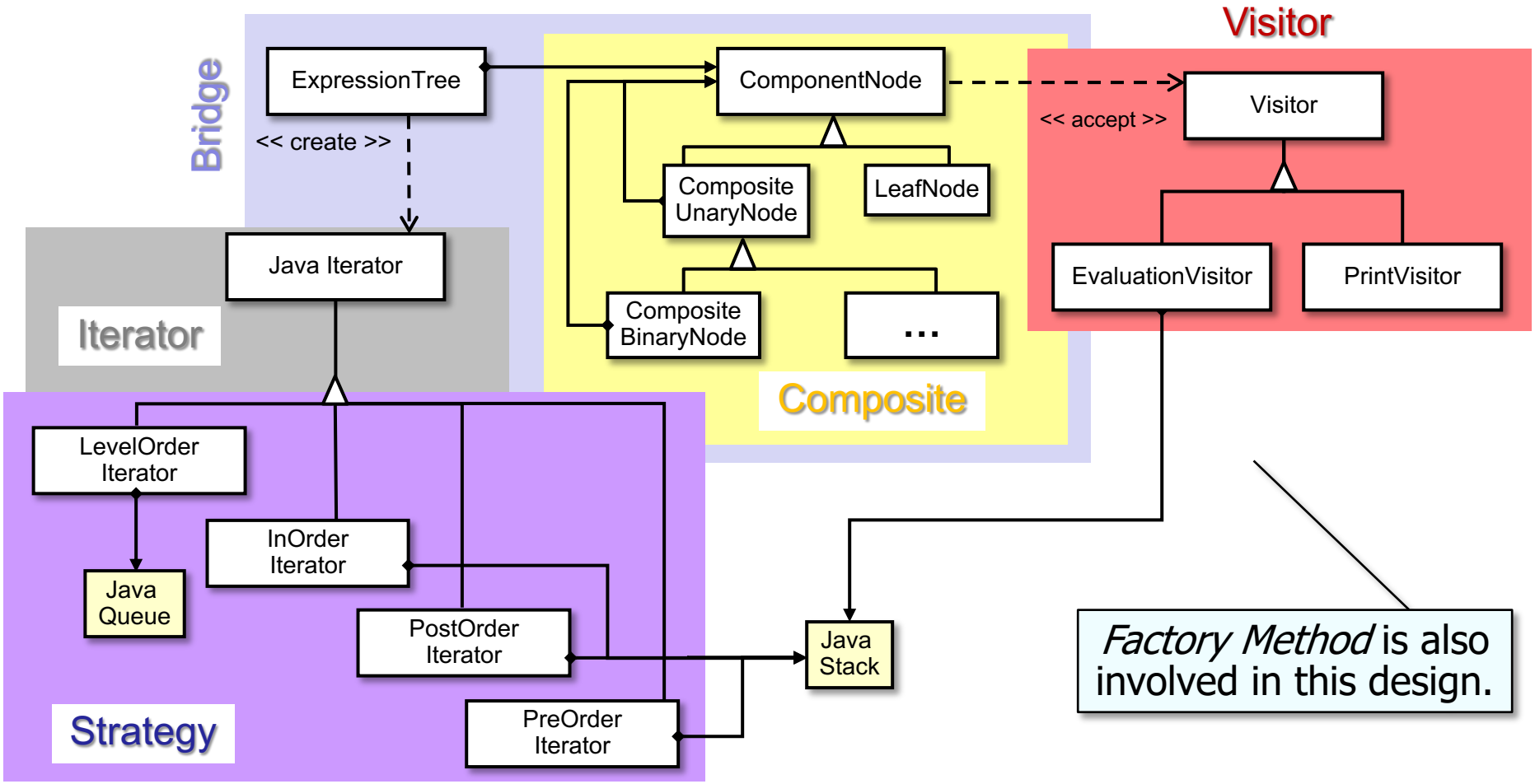
- ProgramNodeEnumerator in Smalltalk-80 compiler
- IRIS Inventor scene rendering
- java.nio.file.FileVisitor and SimpleFileVisitor
- TAO IDL compiler to handle different backends



www.dre.vanderbilt.edu/~schmidt/PDF/ami1.pdf has more information on TAO IDL.

Summary of Visitor Pattern

- *Visitor* works together with *Iterator and Strategy* to traverse the *Composite*-based expression tree flexibly and extensibly perform designated operations.



This pattern sequence simplifies adding new operations without affecting the tree structure.

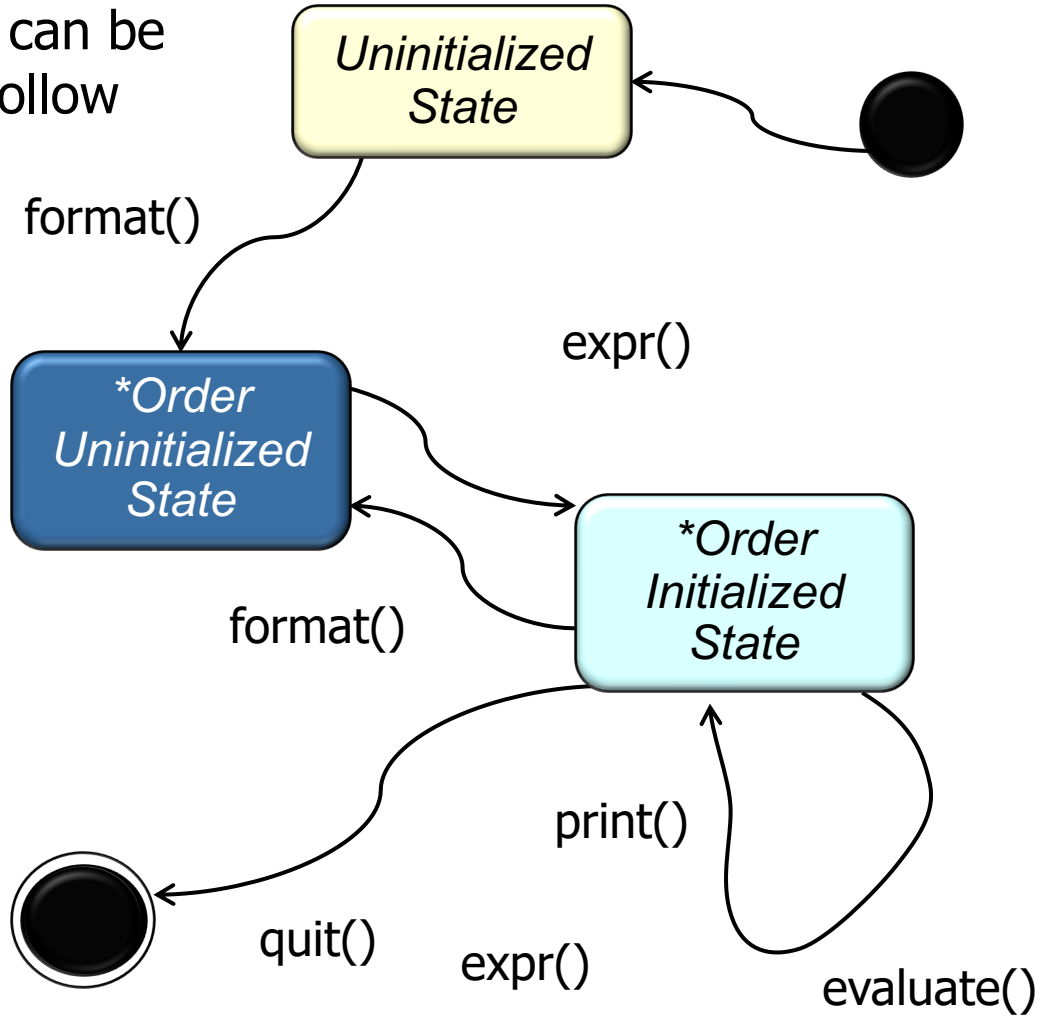
The State Pattern

Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *State* pattern can be applied to ensure user requests follow the correct protocol.



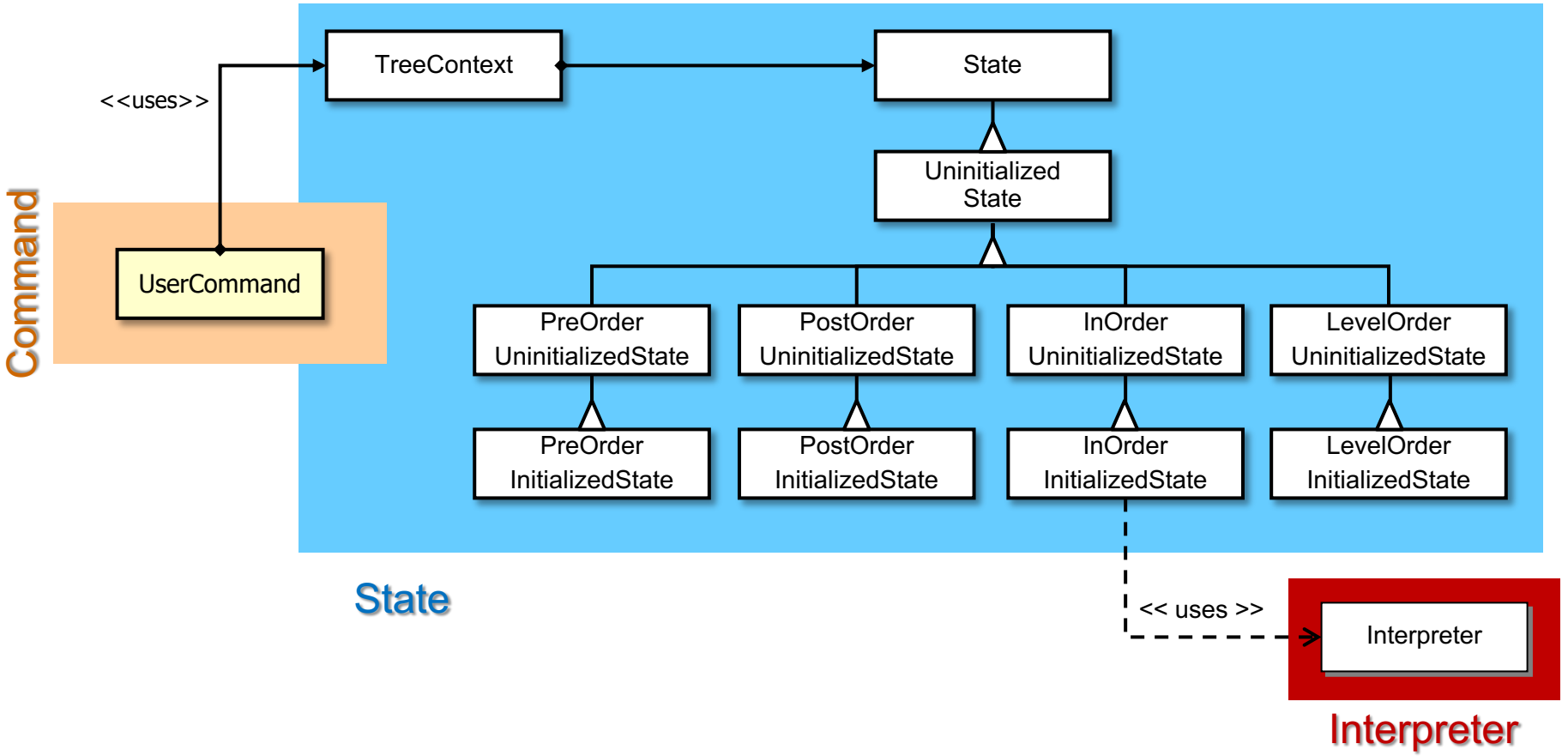
State is one of the most complicated GoF patterns (along with *Visitor*).

Douglas C. Schmidt

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

A Pattern to Enforce User Request Sequencing

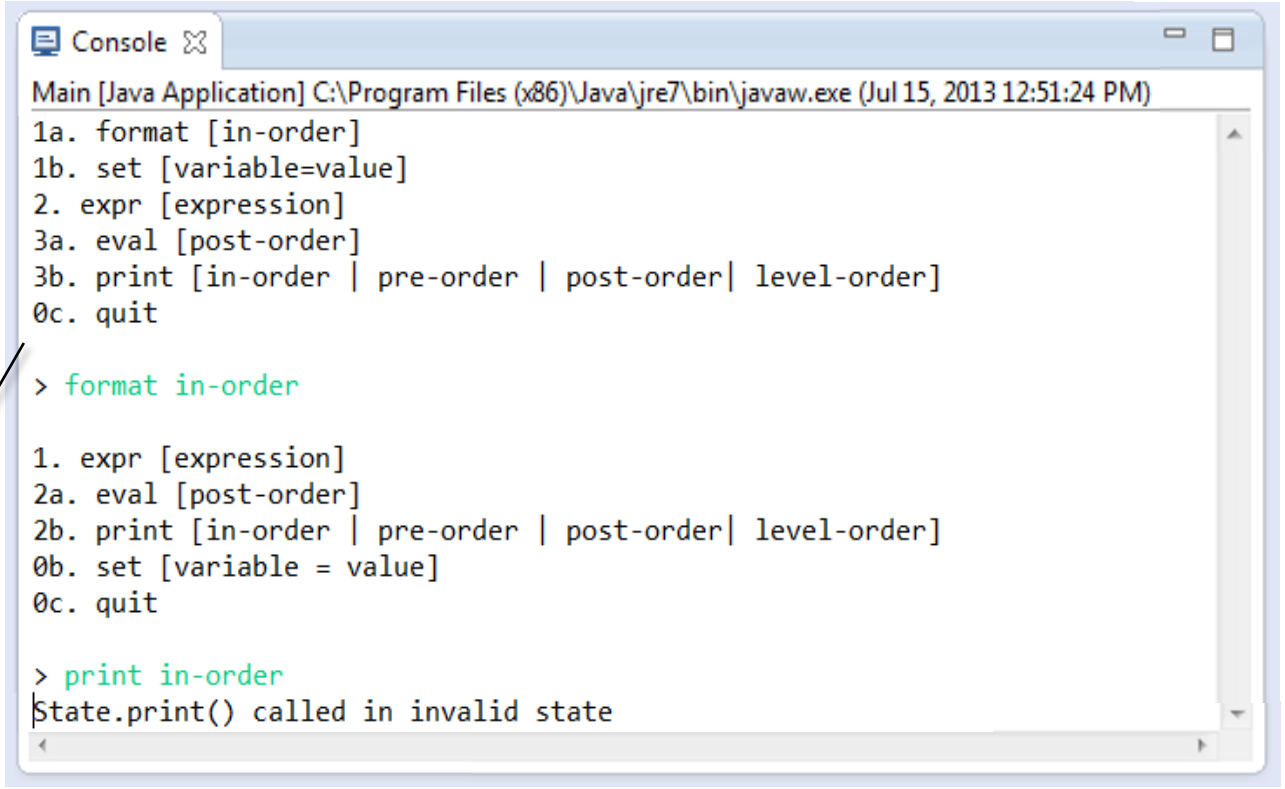
***Purpose**: Ensure the user requests on an expression tree are performed in the correct order*



State structures valid user request sequencing into the class hierarchy design.

Context: OO Expression Tree Processing App

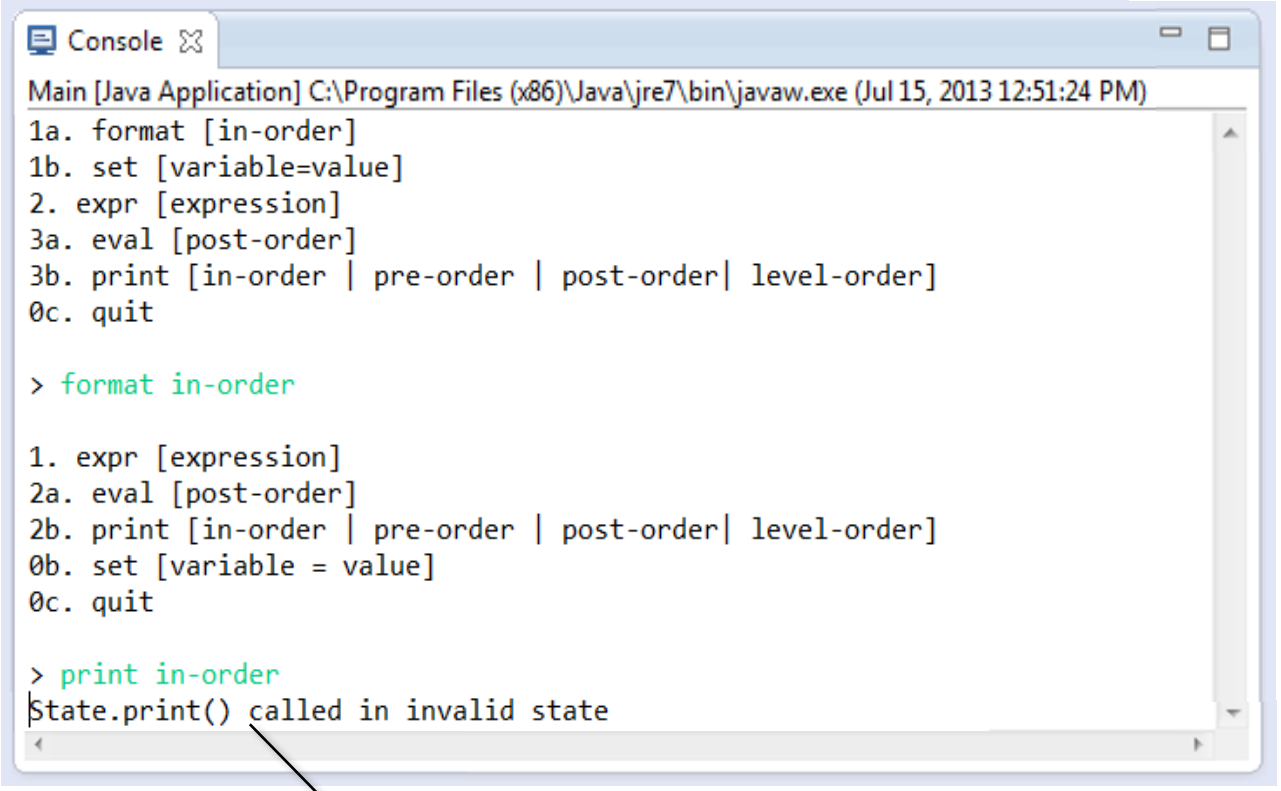
- Users must follow the correct protocol when making requests on an expression tree



This example demonstrates "verbose mode."

Context: OO Expression Tree Processing App

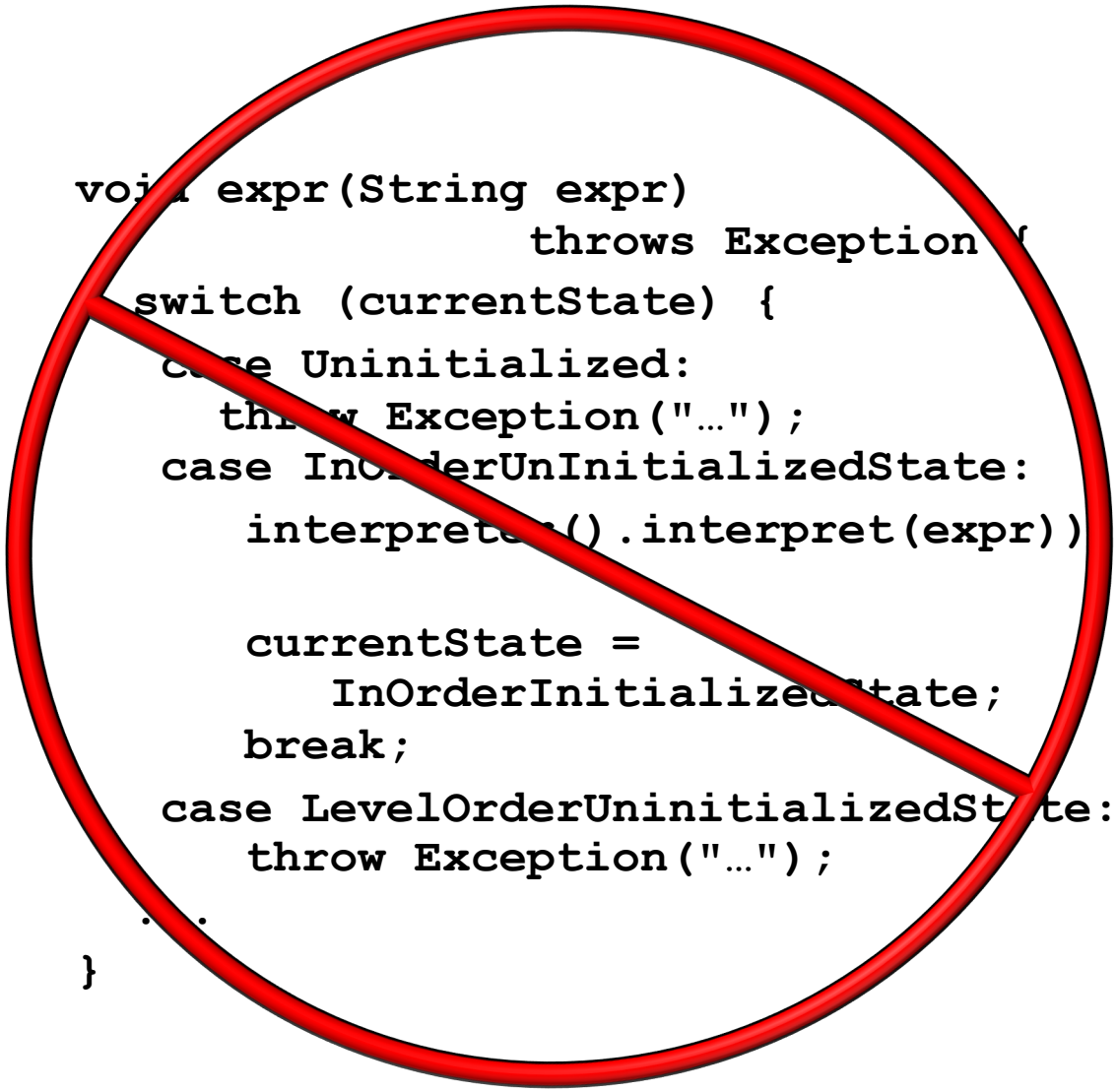
- Users must follow the correct protocol when making requests on an expression tree, e.g.,
  1. **`format`** must be called first.
  2. **`expr`** must be called before **`print`** or **`eval`**.
  3. **`print`** and **`eval`** can be called in any order after **`expr`** is called.
  4. Etc.



*Protocol violation since **`print`** was called before **`expr`** was called*

Problem: Incorrect User Request Ordering

- Large monolithic if/else conditional statements or switch statements are hard to write and maintain.

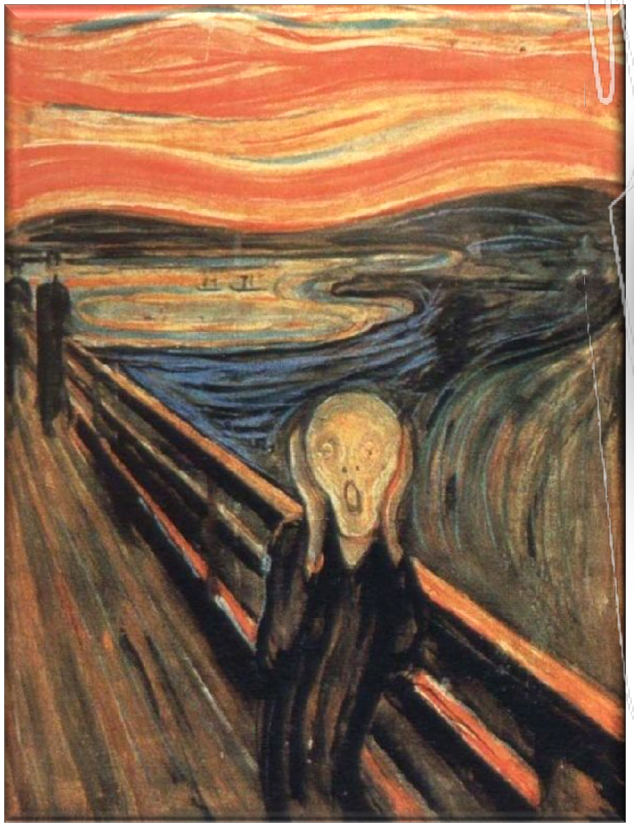


Problem: Incorrect User Request Ordering

- Large monolithic if/else conditional statements or switch statements are hard to write and maintain.
  - The logic of the state machine is tightly coupled with the functionality of the code.

Problem: Incorrect User Request Ordering

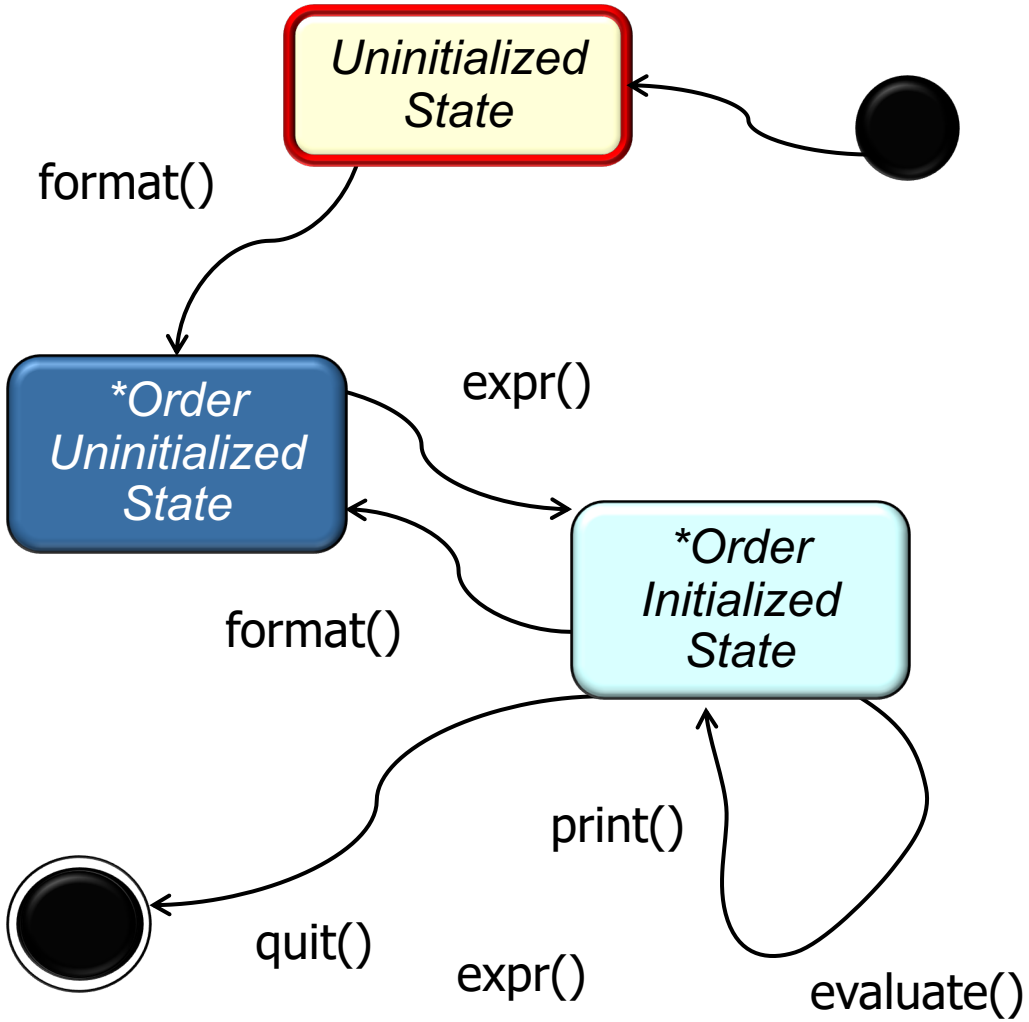
- Large monolithic if/else conditional statements or switch statements are hard to write and maintain.
  - The logic of the state machine is tightly coupled with the functionality of the code.



Writing this spaghetti code for each user request is tedious and error-prone.

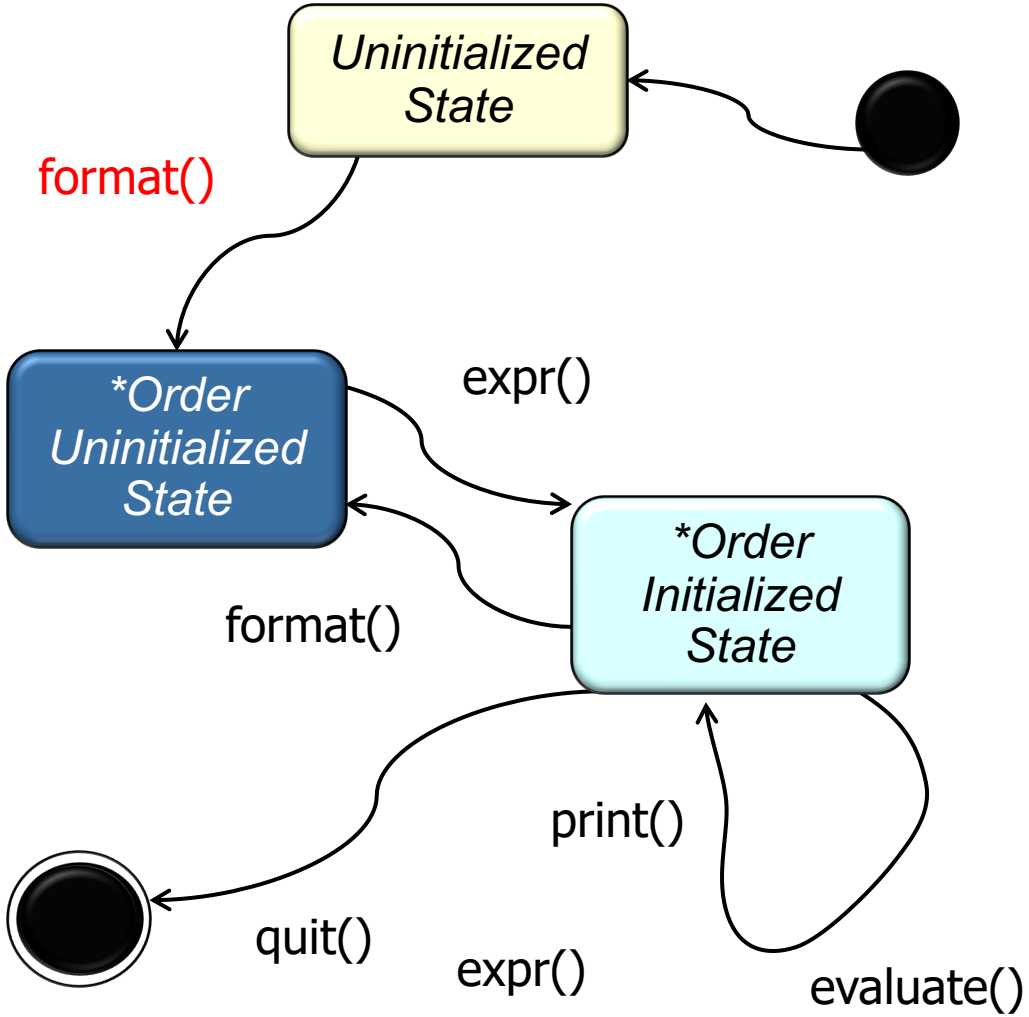
Solution: Encapsulate Request History as States

- Define an object whose behavior depends on its state.
  - Valid sequences of user requests can be represented via a state machine.



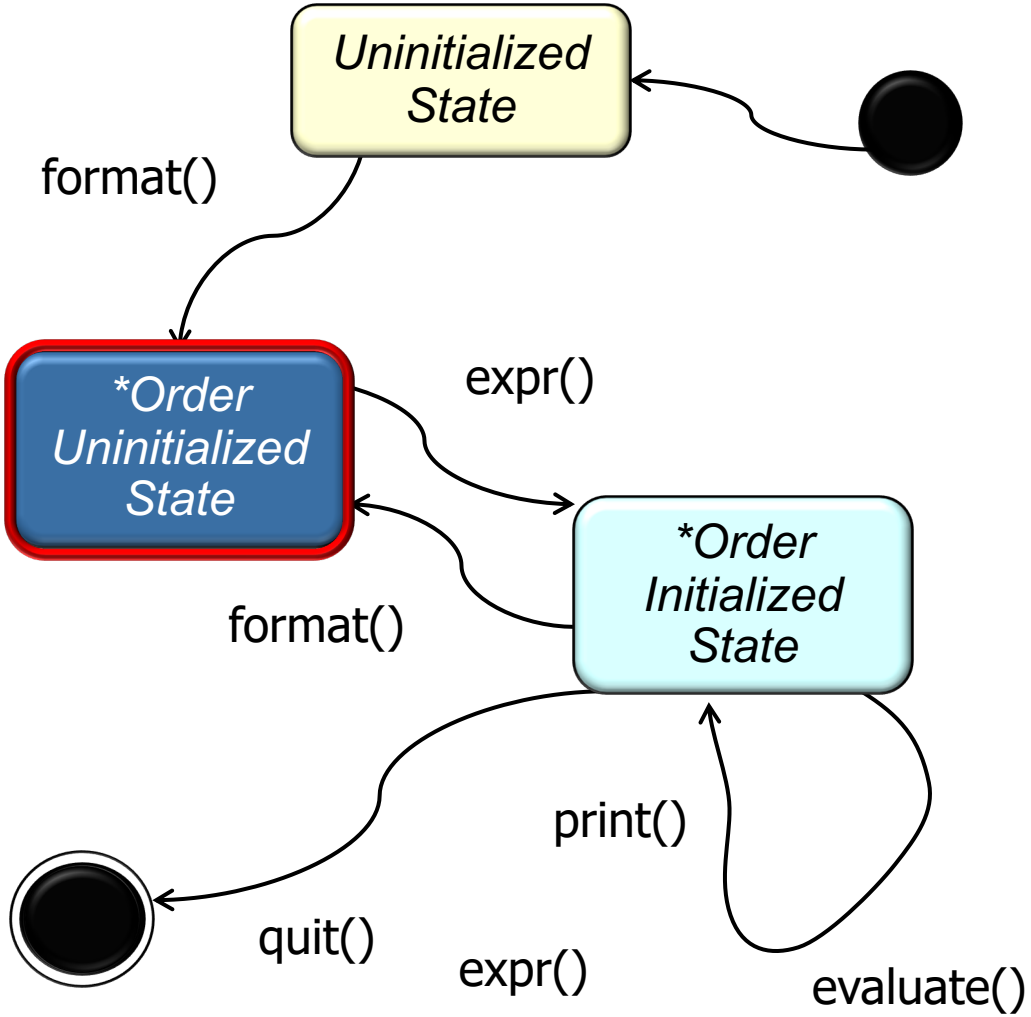
Solution: Encapsulate Request History as States

- Define an object whose behavior depends on its state.
  - Valid sequences of user requests can be represented via a state machine.



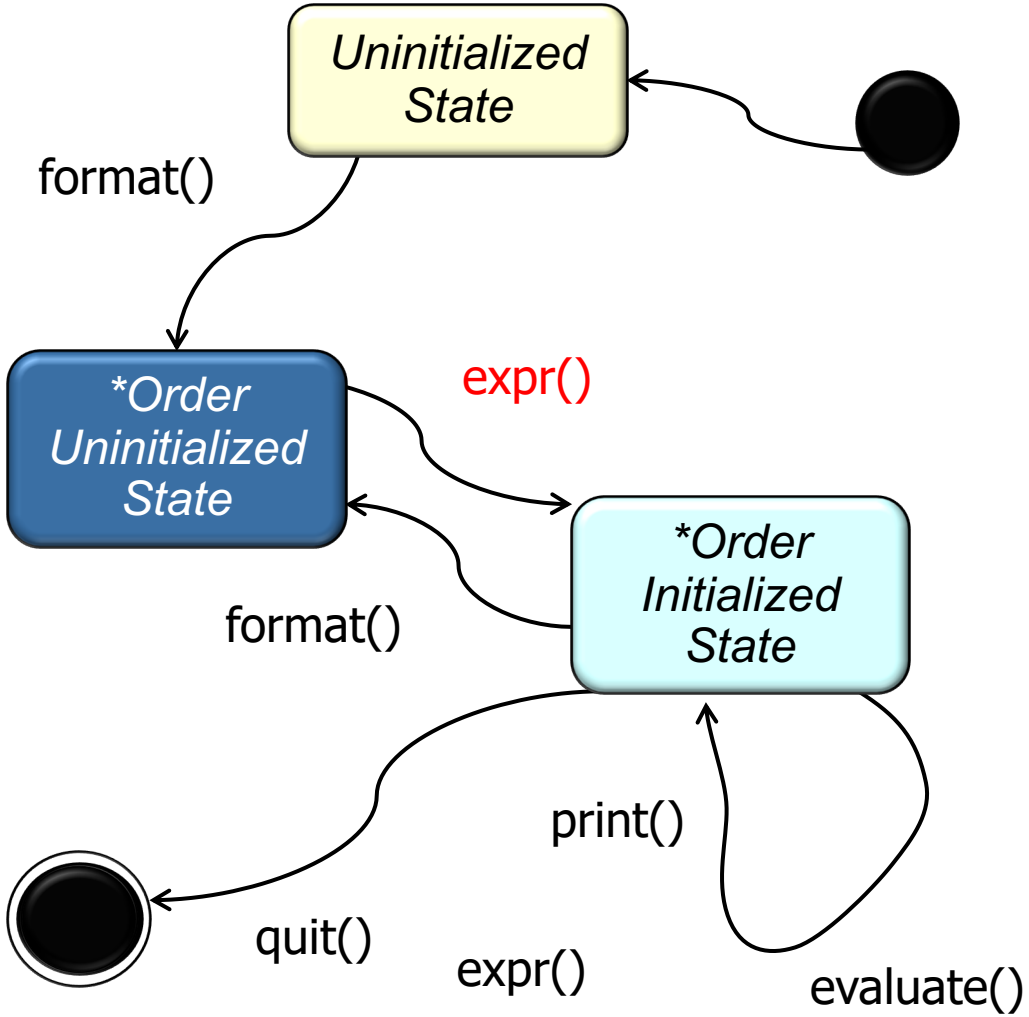
Solution: Encapsulate Request History as States

- Define an object whose behavior depends on its state.
  - Valid sequences of user requests can be represented via a state machine.



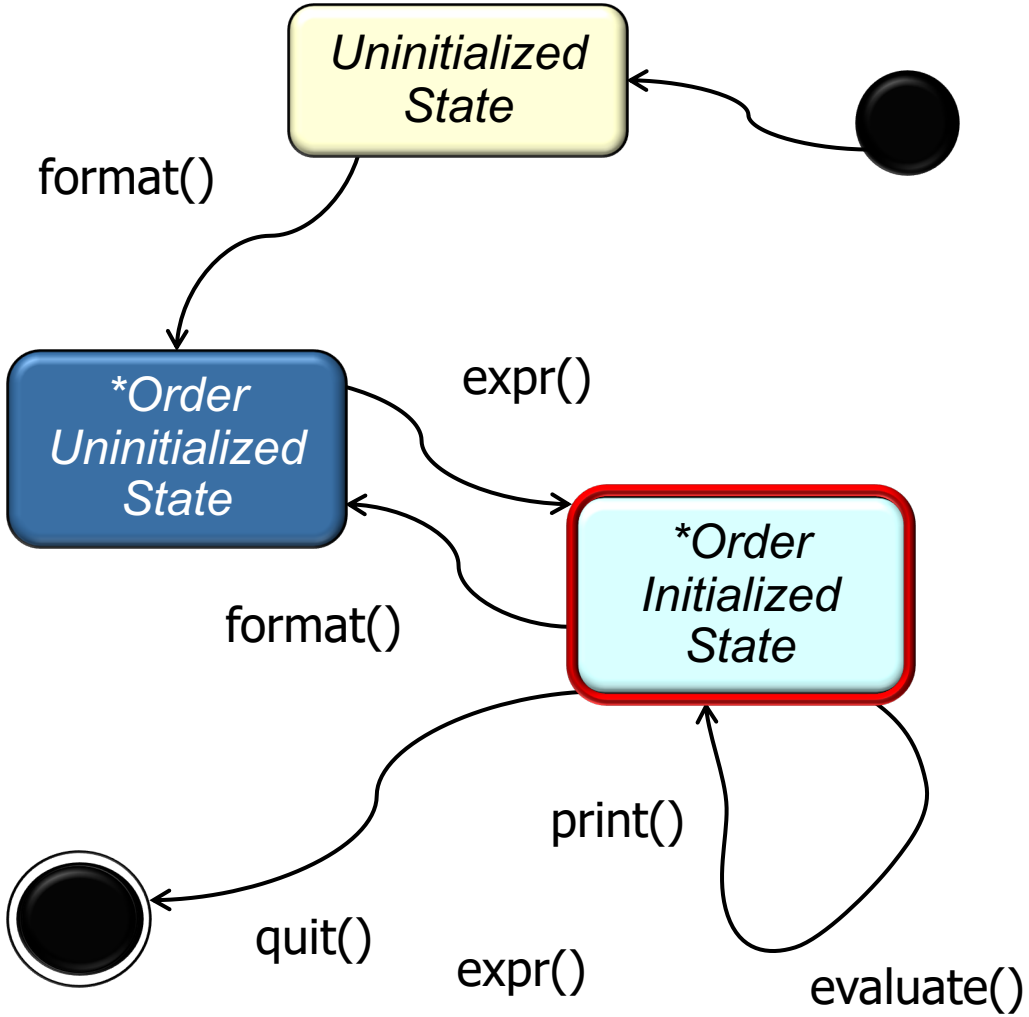
Solution: Encapsulate Request History as States

- Define an object whose behavior depends on its state.
  - Valid sequences of user requests can be represented via a state machine.



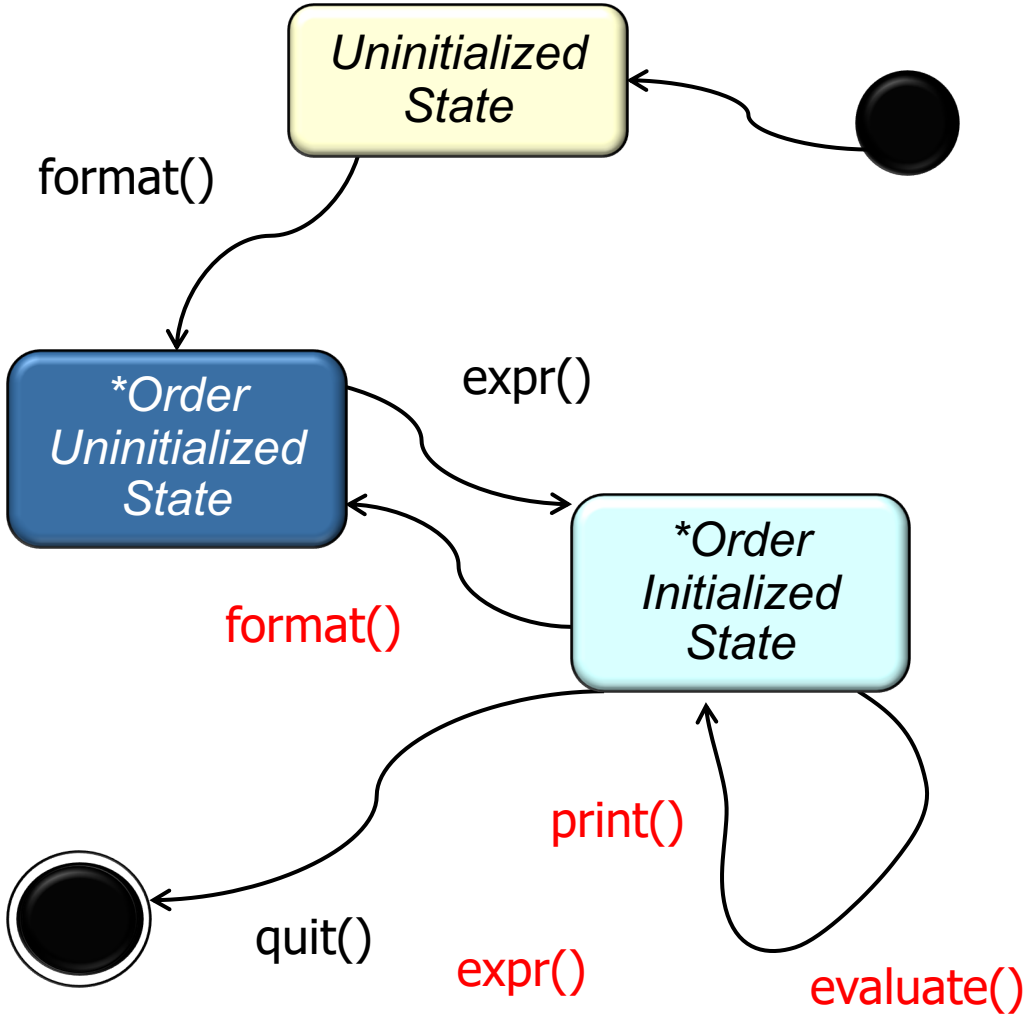
Solution: Encapsulate Request History as States

- Define an object whose behavior depends on its state.
  - Valid sequences of user requests can be represented via a state machine.



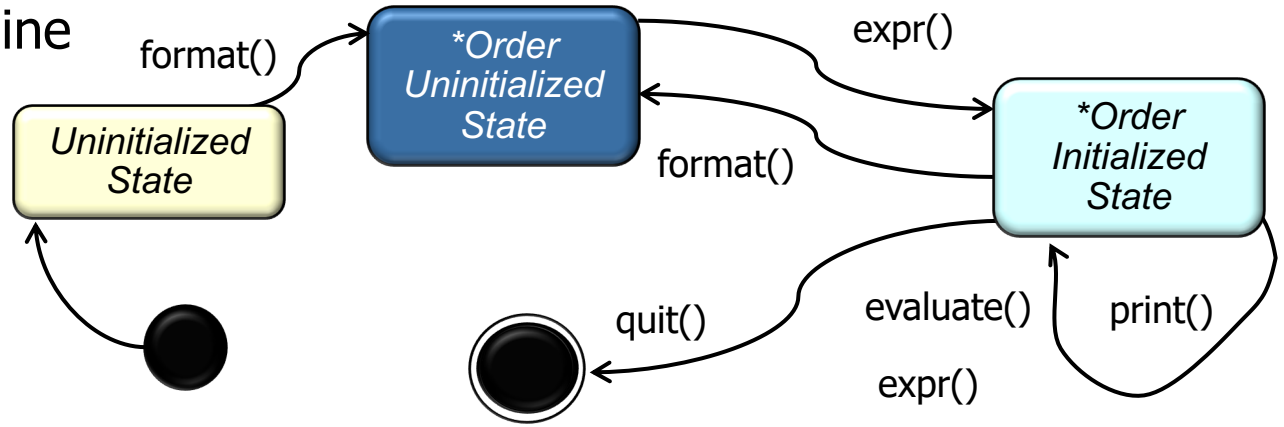
Solution: Encapsulate Request History as States

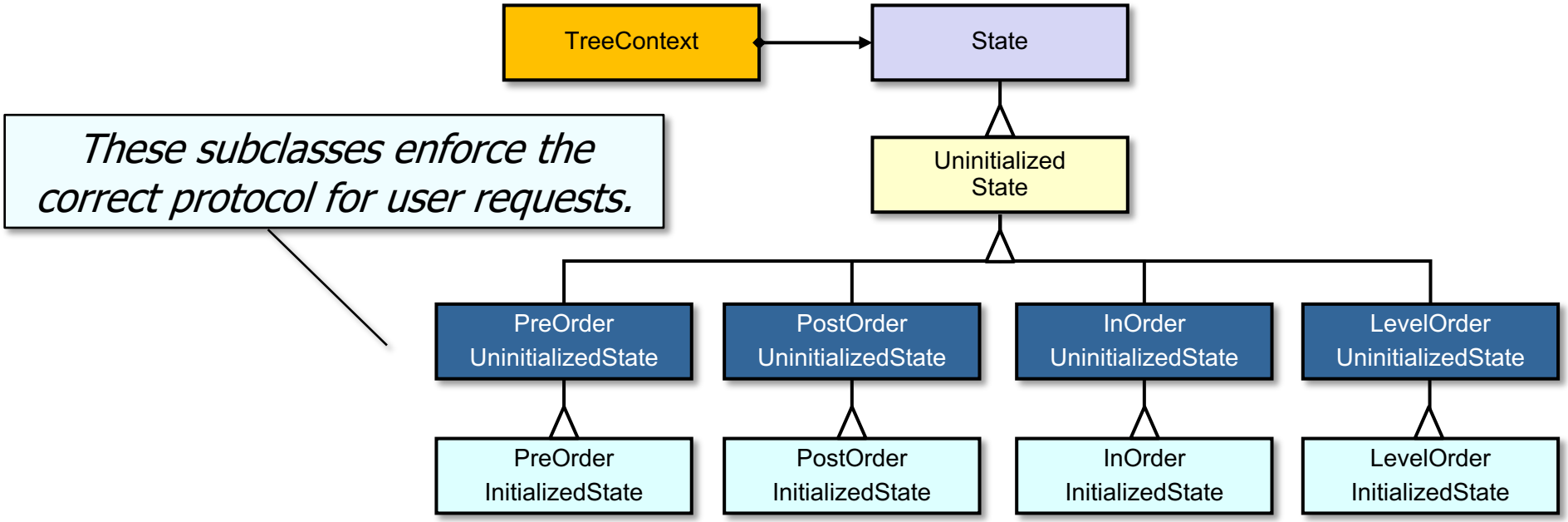
- Define an object whose behavior depends on its state.
  - Valid sequences of user requests can be represented via a state machine.



Solution: Encapsulate Request History as States

- Encode the state machine using subclasses.





TreeContext Class Overview

- Context used to ensure user requests are performed according to protocol

Class methods

```
          void  format(String newformat)
          void  expr(String expression)
          void  print(String format)
          void  evaluate(String format)

         State  state()
          void  state(State newstate)
ExpressionTree  tree()
          void  tree(ExpressionTree newtree)
```

`TreeContext` encapsulates variability via a "closed" interface, a la *Bridge.*

TreeContext Class Overview

- Context used to ensure user requests are performed according to protocol

Class methods

These methods correspond to user commands

```
          void  format(String newformat)
          void  expr(String expression)
          void  print(String format)
          void  evaluate(String format)

         State  state()
          void  state(State newstate)
ExpressionTree  tree()
          void  tree(ExpressionTree newtree)
```

TreeContext Class Overview

- Context used to ensure user requests are performed according to protocol

Class methods

```
          void  format(String newformat)
          void  expr(String expression)
          void  print(String format)
          void  evaluate(String format)
```

Setter/getter for `State` subclasses

```
         State  state()
          void  state(State newstate)
ExpressionTree  tree()
          void  tree(ExpressionTree newtree)
```

TreeContext Class Overview

- Context used to ensure user requests are performed according to protocol

Class methods

```
          void  format(String newformat)
          void  expr(String expression)
          void  print(String format)
          void  evaluate(String format)

         State  state()
          void  state(State newstate)
ExpressionTree  tree()
          void  tree(ExpressionTree newtree)
```

Setter/getter for `ExpressionTree`

TreeContext Class Overview

- Context used to ensure user requests are performed according to protocol

Class methods

```
          void  format(String newformat)
          void  expr(String expression)
          void  print(String format)
          void  evaluate(String format)

         State  state()
          void  state(State newstate)
ExpressionTree  tree()
          void  tree(ExpressionTree newtree)
```

- **Commonality**: provides a common set of methods for performing user requests
- **Variability**: each method delegates to the appropriate subclass of `State`, depending on the state of the context

State Class Hierarchy Overview

- Inheritance hierarchy that defines subclasses for each state to determine how users' requests are processed

Super class methods

```
void  format(TreeContext opsContext, String newformat)
void  expr(TreeContext opsContext, String expression)
void  print(TreeContext opsContext, String format)
void  evaluate(TreeContext opsContext, String format)
```

State Class Hierarchy Overview

- Inheritance hierarchy that defines subclasses for each state to determine how users' requests are processed

Super class methods

```
void  format(TreeContext opsContext, String newformat)
void  expr(TreeContext opsContext, String expression)
void  print(TreeContext opsContext, String format)
void  evaluate(TreeContext opsContext, String format)
```

`TreeContext` methods delegate to `State` methods by including the original `TreeContext` object as an extra argument to each delegated method.

See "Delegation in C++" at www.dre.vanderbilt.edu/~schmidt/PDF/joop0407.pdf

State Class Hierarchy Overview

- Inheritance hierarchy that defines subclasses for each state to determine how users' requests are processed

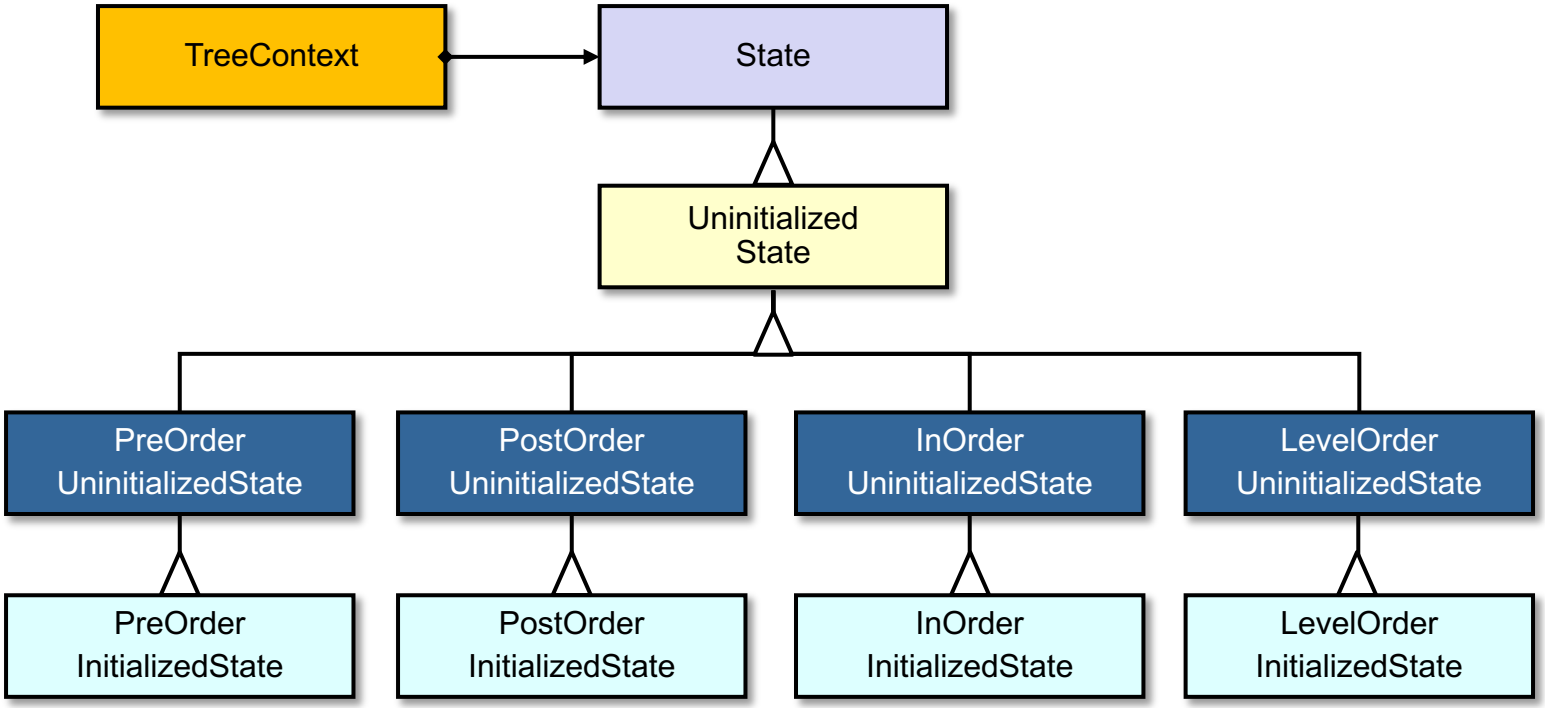
Super class methods

```
void  format(TreeContext opsContext, String newformat)
void  expr(TreeContext opsContext, String expression)
void  print(TreeContext opsContext, String format)
void  evaluate(TreeContext opsContext, String format)
```

- **Commonality**: provides a common set of methods for performing user requests
- **Variability**: each subclass of `State` can implement these methods in a different way, depending on the current state

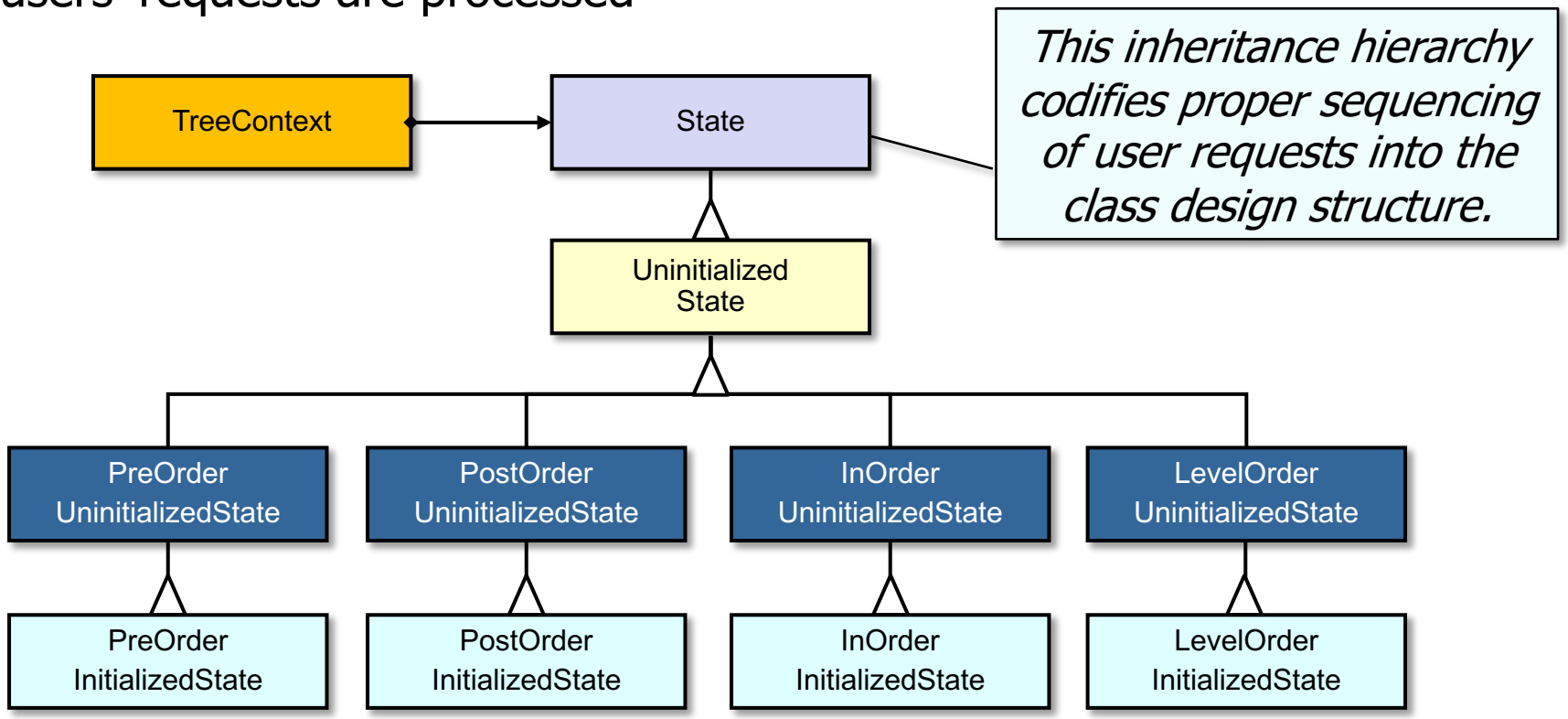
State Class Hierarchy Overview

- Inheritance hierarchy that defines subclasses for each state to determine how users' requests are processed



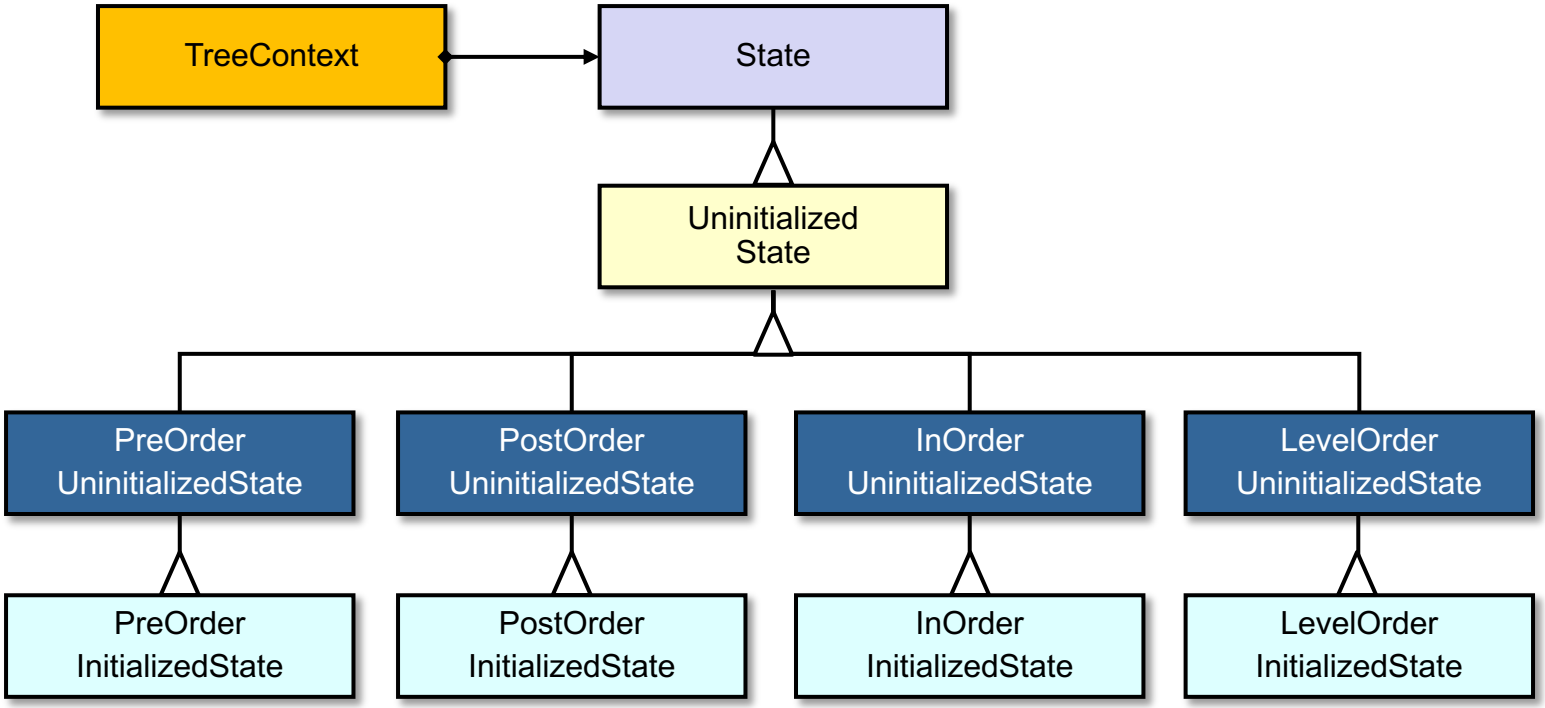
State Class Hierarchy Overview

- Inheritance hierarchy that defines subclasses for each state to determine how users' requests are processed



State Class Hierarchy Overview

- Inheritance hierarchy that defines subclasses for each state to determine how users' requests are processed



- **Commonality**: provides a common interface for ensuring that expression tree requests are invoked by users according to the correct protocol
- **Variability**: the implementations—and correctness—of the expression tree requests vary depending on the user request and the current state

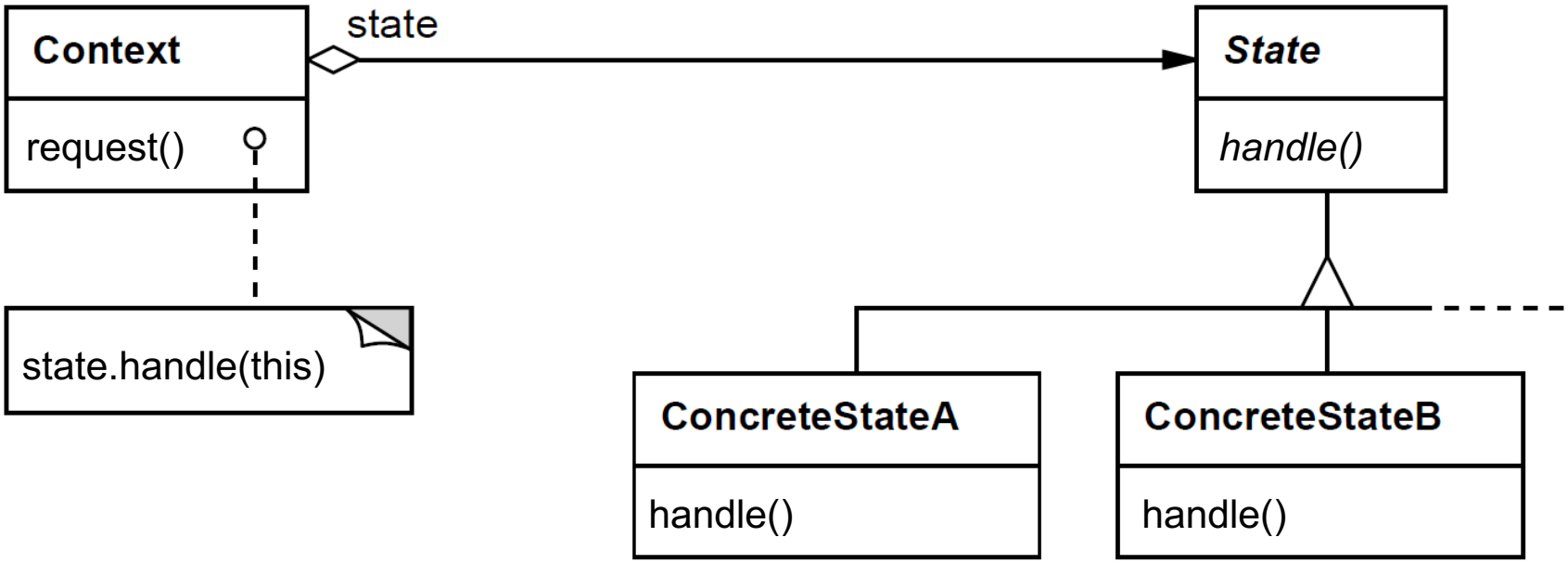
The State Pattern

Structure and Functionality

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *State* pattern can be applied to ensure user requests follow the correct protocol.
- Understand the structure and functionality of the *State* pattern.

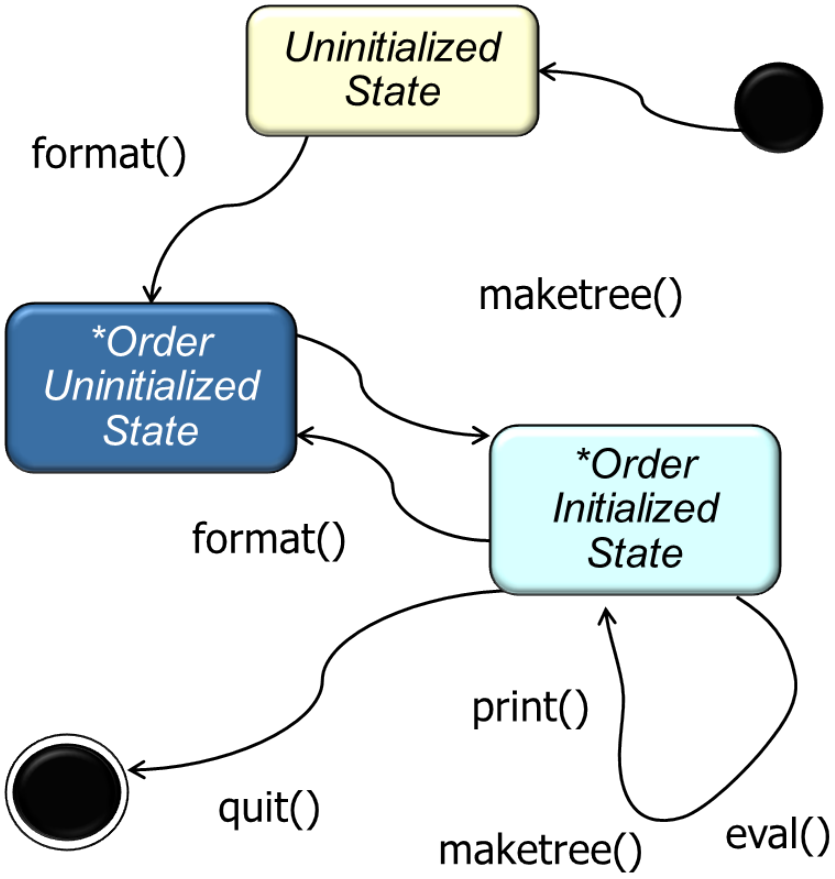


Douglas C. Schmidt

Structure and Functionality of the State Pattern

Intent

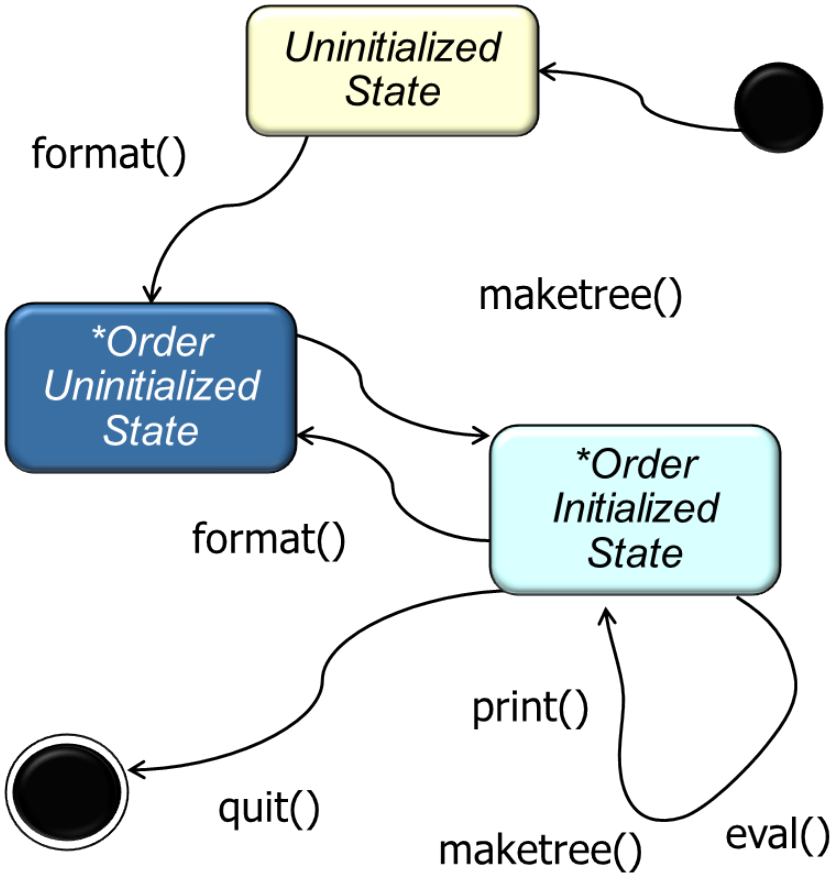
- Allow an object to alter its behavior when its internal state changes



See en.wikipedia.org/wiki/State_pattern

Intent

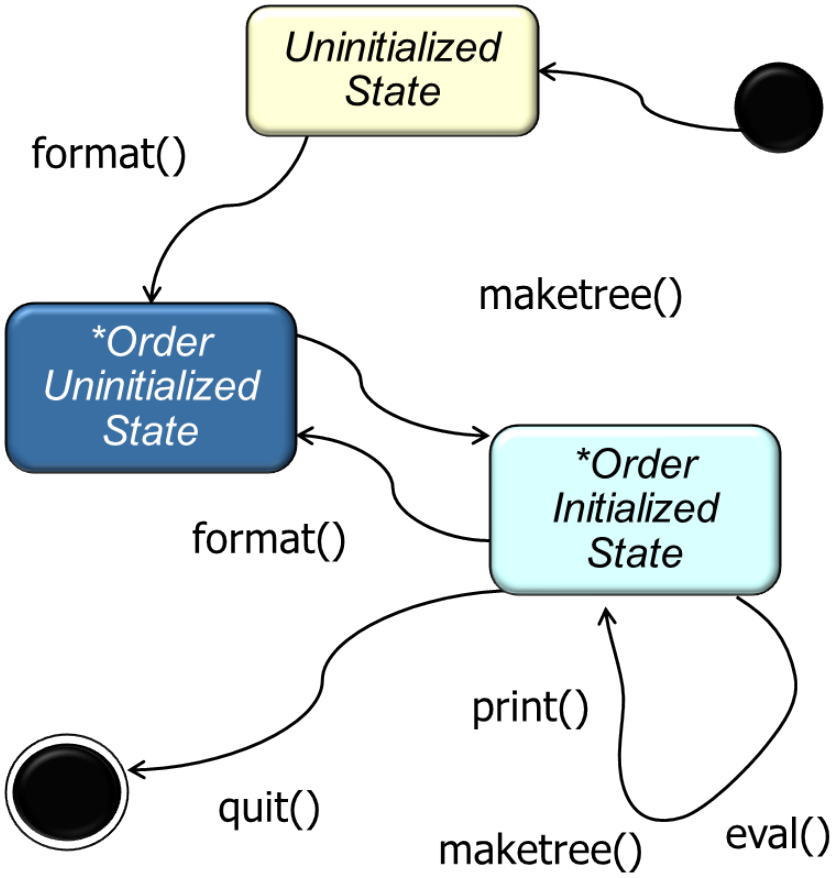
- Allow an object to alter its behavior when its internal state changes, i.e.,
  - The object will appear to change its class *without* changing its API!



"Change its class" == "its (same) methods behave differently"

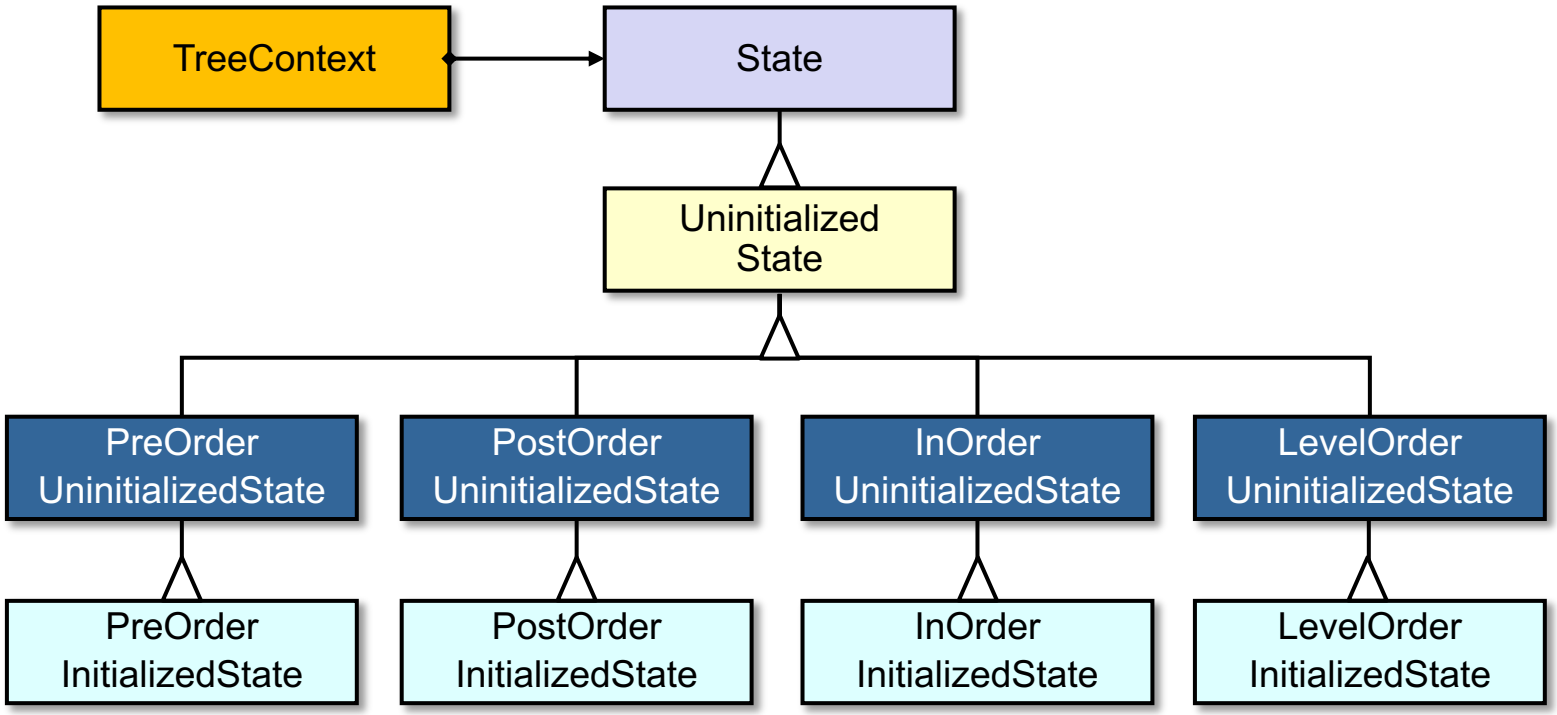
Applicability

- When an object must change its behavior at runtime depending on which state it is in

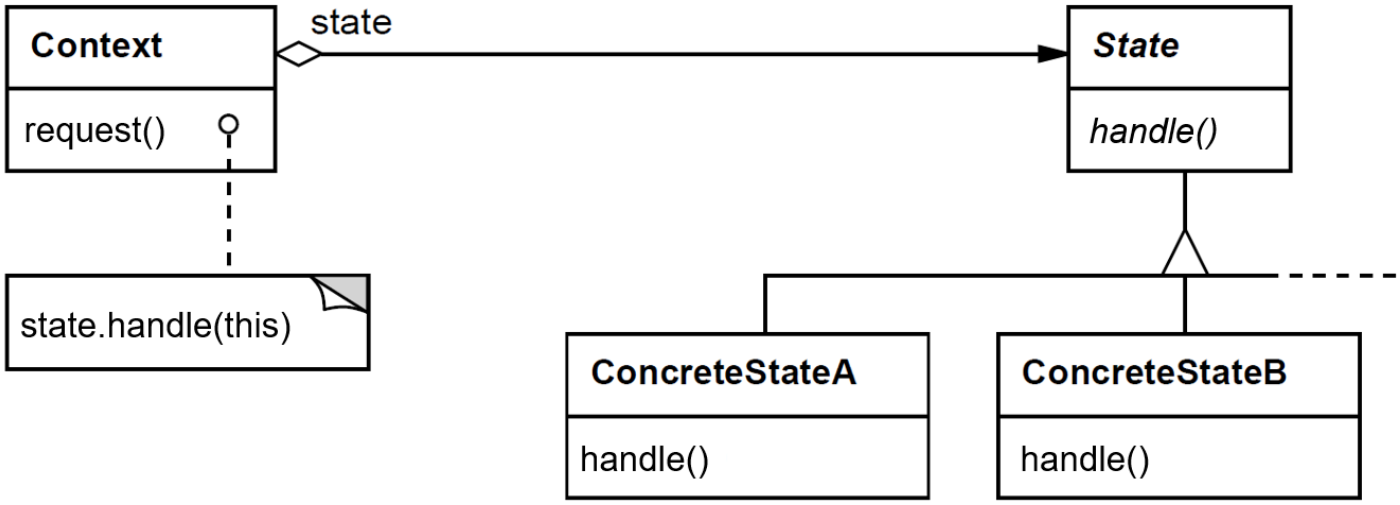


Applicability

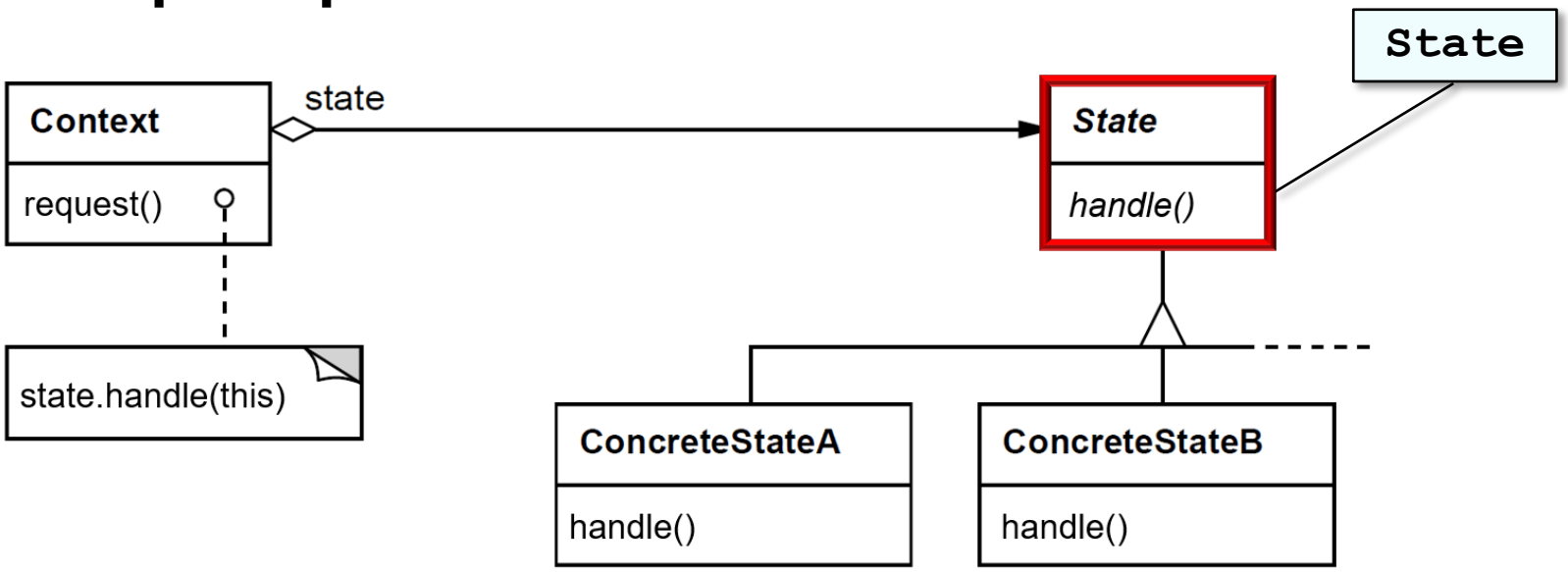
- When an object must change its behavior at runtime depending on which state it is in
- When several operations have the same large multi-part conditional structure that depends on the object's state



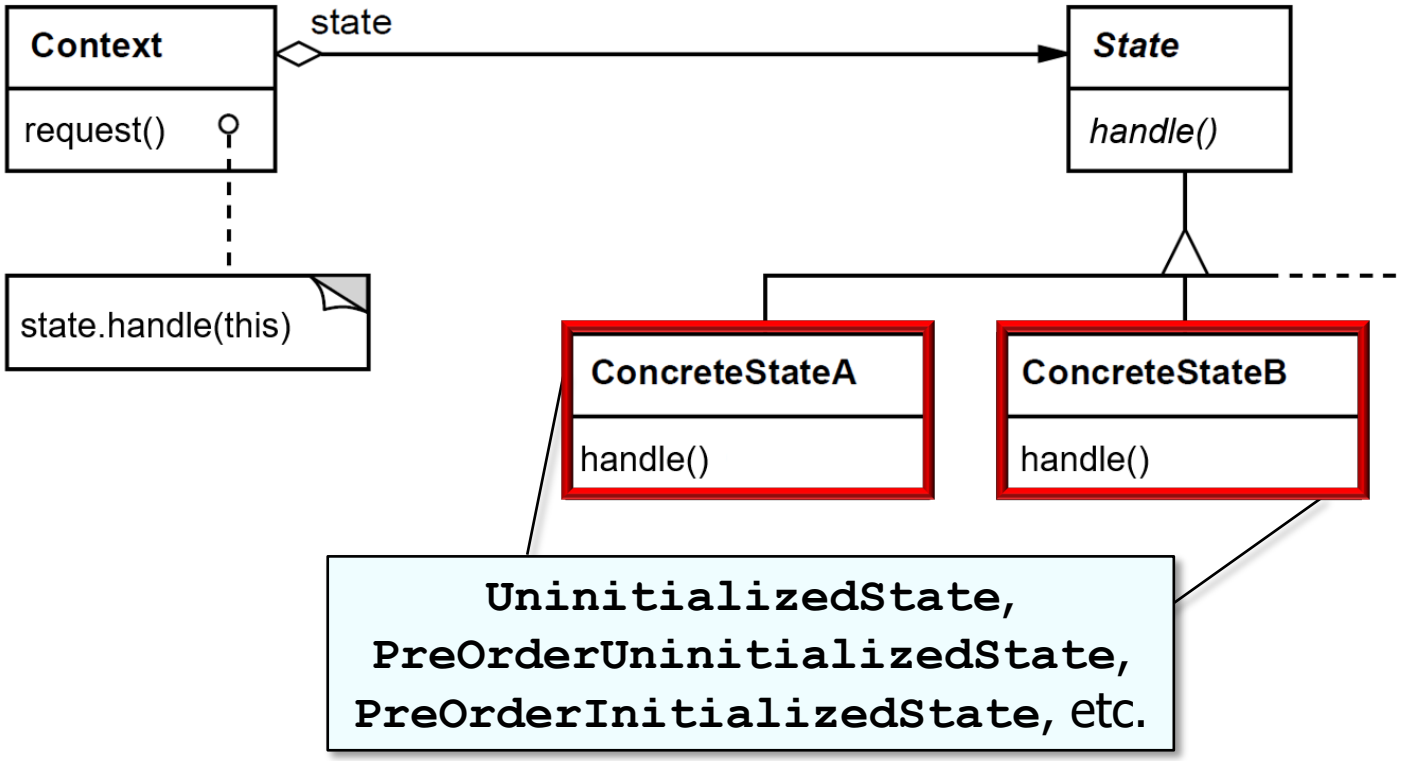
Structure and participants



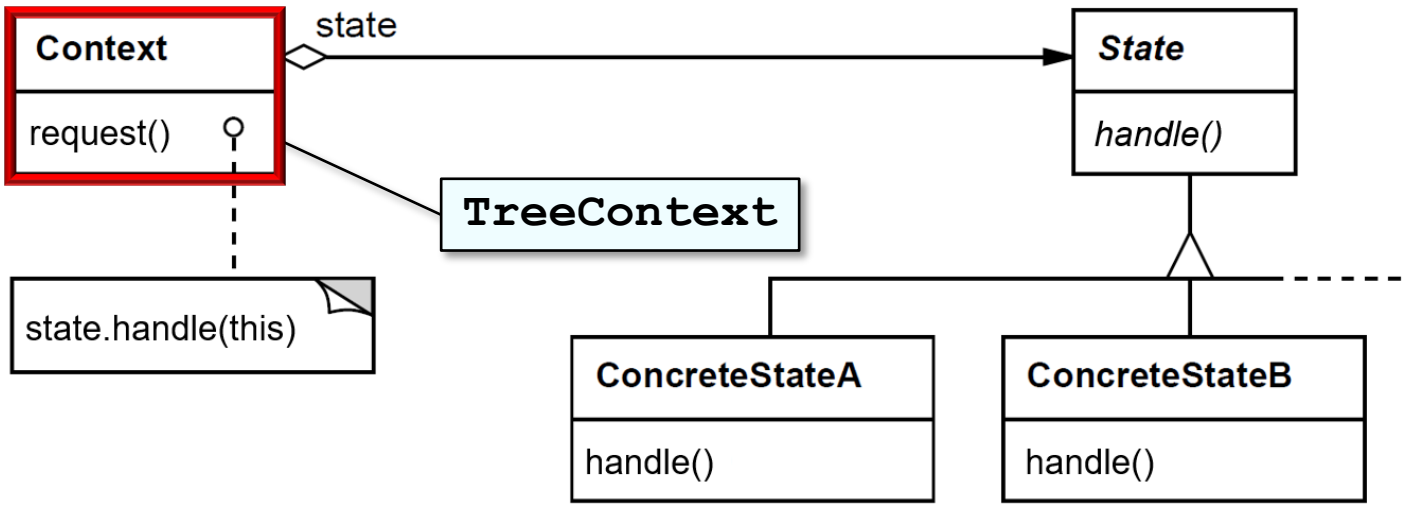
Structure and participants



Structure and participants

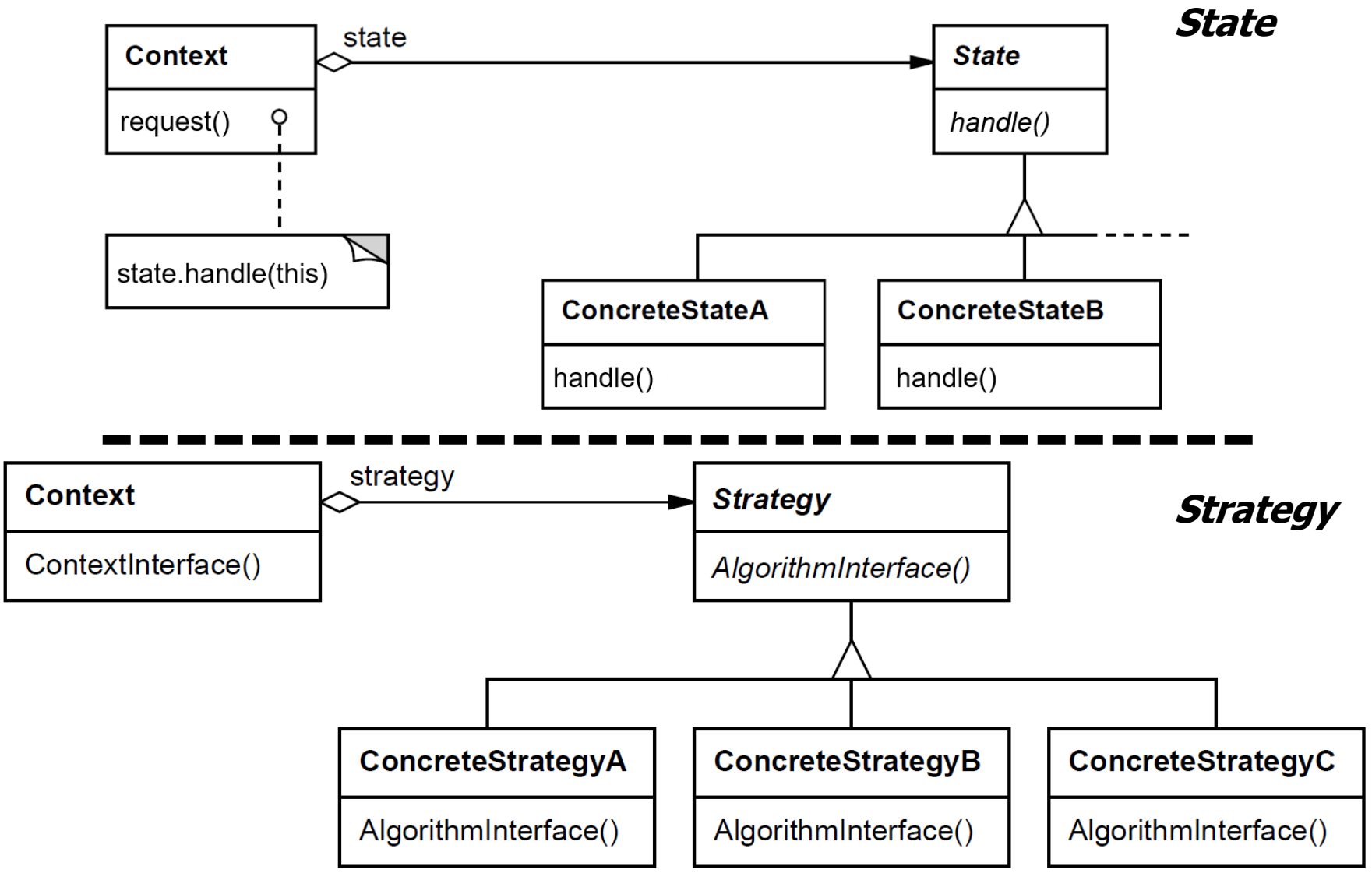


Structure and participants



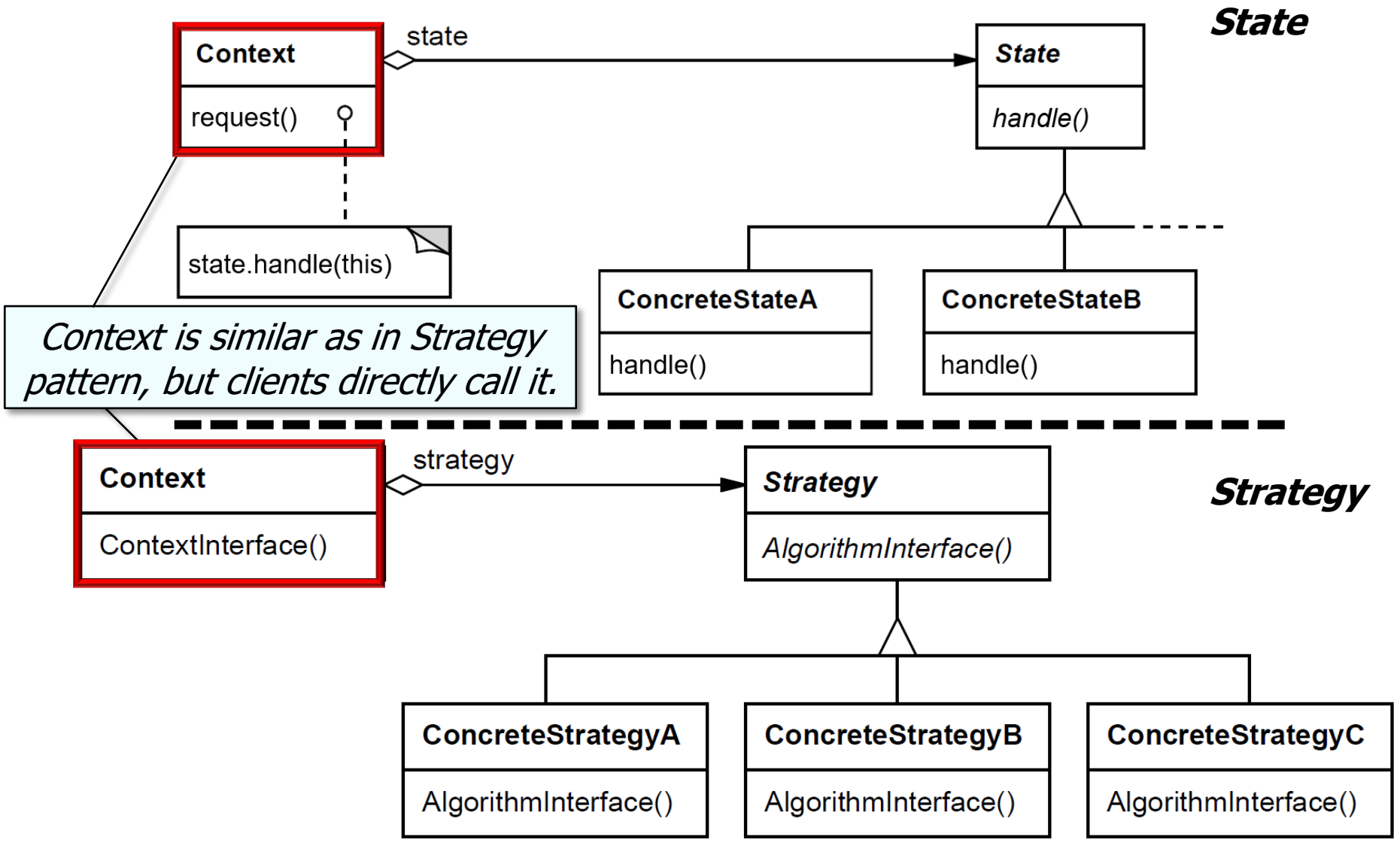
Clients invoke requests via the Context object.

Structure and participants



State is like *Strategy* where the strategy can change via invocations on context.

Structure and participants



The **`Context`** in *State* is similar to the **`Abstraction`** in *Bridge.*

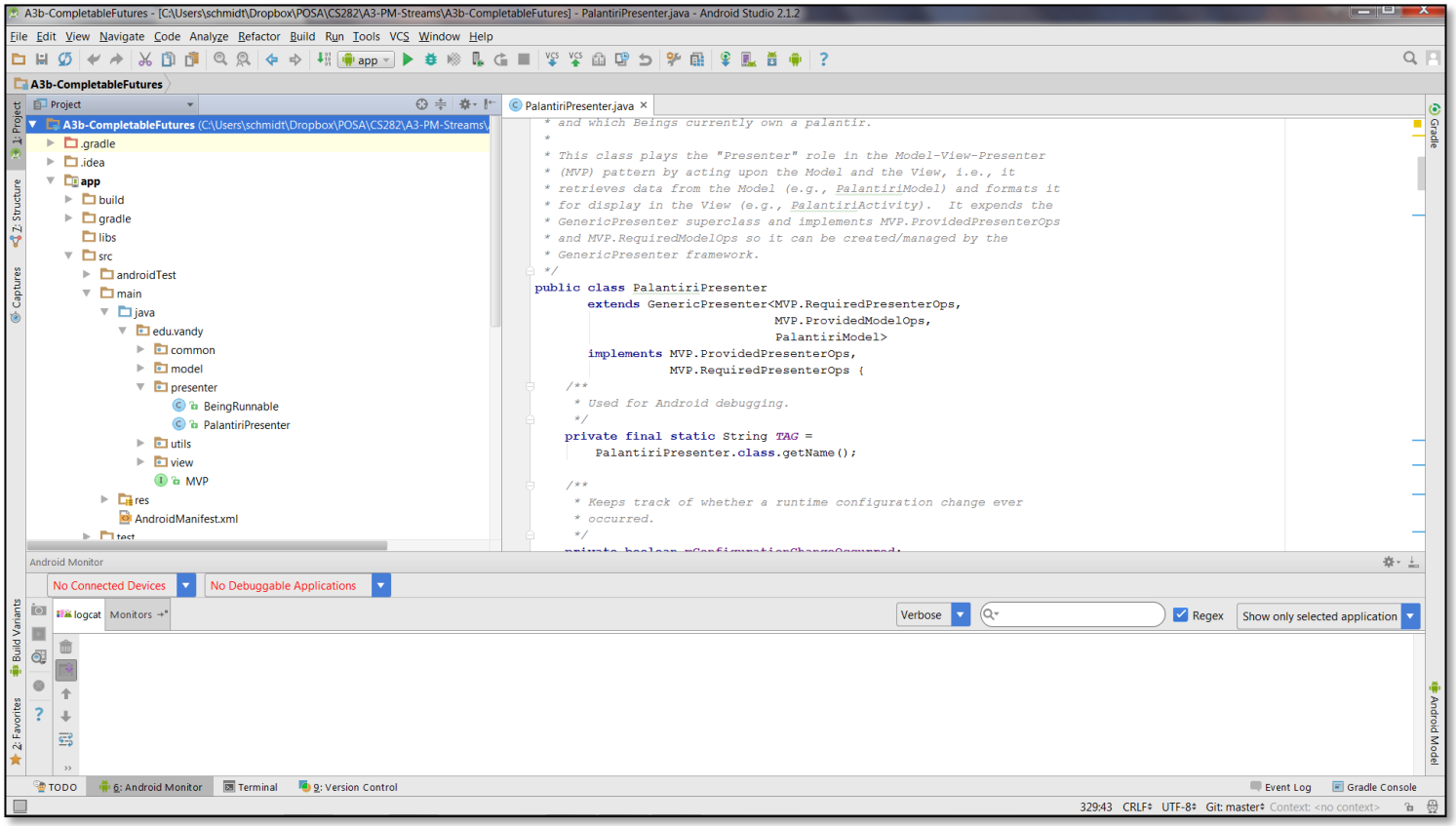
The State Pattern

Implementation in Java

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *State* pattern can be applied to ensure user requests follow the correct protocol.
- Understand the structure and functionality of the *State* pattern.
- Know how to implement the *State* pattern in Java.



State example in Java

- Allows the **`TreeContext`** object to alter its behavior when its state changes

```
public class TreeContext {
  void expr(String expression)
  { mState.expr(this, expression); } ...
```

See ExpressionTree/CommandLine/src/expressiontree/states

State example in Java

- Allows the **`TreeContext`** object to alter its behavior when its state changes

```
public class TreeContext {
  void expr(String expression)
  { mState.expr(this, expression); } ...
```

This method delegates to the `State` object

State example in Java

- Allows the **`TreeContext`** object to alter its behavior when its state changes

```
public class TreeContext {
  void expr(String expression)
  { mState.expr(this, expression); } ...

class UninitializedState extends State {
  void expr(TreeContext context, String expression) {
    throw new IllegalStateException("called in invalid state");
  } ...
```

It's invalid to call expr() in this state since the user hasn't provided an input format yet.

State example in Java

- Allows the **`TreeContext`** object to alter its behavior when its state changes

```
public class TreeContext {
  void expr(String expression)
  { mState.expr(this, expression); } ...

class UninitializedState extends State {
  void expr(TreeContext context, String expression) {
    throw new IllegalStateException("called in invalid state");
  } ...

class InOrderUninitializedState extends UninitializedState {
```

Calling `expr()` in this state uses *Interpreter* to build the expression tree.

```
  void expr(TreeContext context, String expression) {
    context.tree(context.mInterpreter.interpret(expression));
    context.state(new InOrderInitializedState());
  } ...
```

State example in Java

- Allows the **`TreeContext`** object to alter its behavior when its state changes

```
public class TreeContext {
  void expr(String expression)
  { mState.expr(this, expression); } ...

class UninitializedState extends State {
  void expr(TreeContext context, String expression) {
    throw new IllegalStateException("called in invalid state");
  } ...

class InOrderUninitializedState extends UninitializedState {


  void expr(TreeContext context, String expression) {
    context.tree(context.mInterpreter.interpret(expression));
    context.state(new InOrderInitializedState());
  } ...
```

Transition to next state

It's easy to change the internal state since the context is delegated!

The State Pattern

Other Considerations

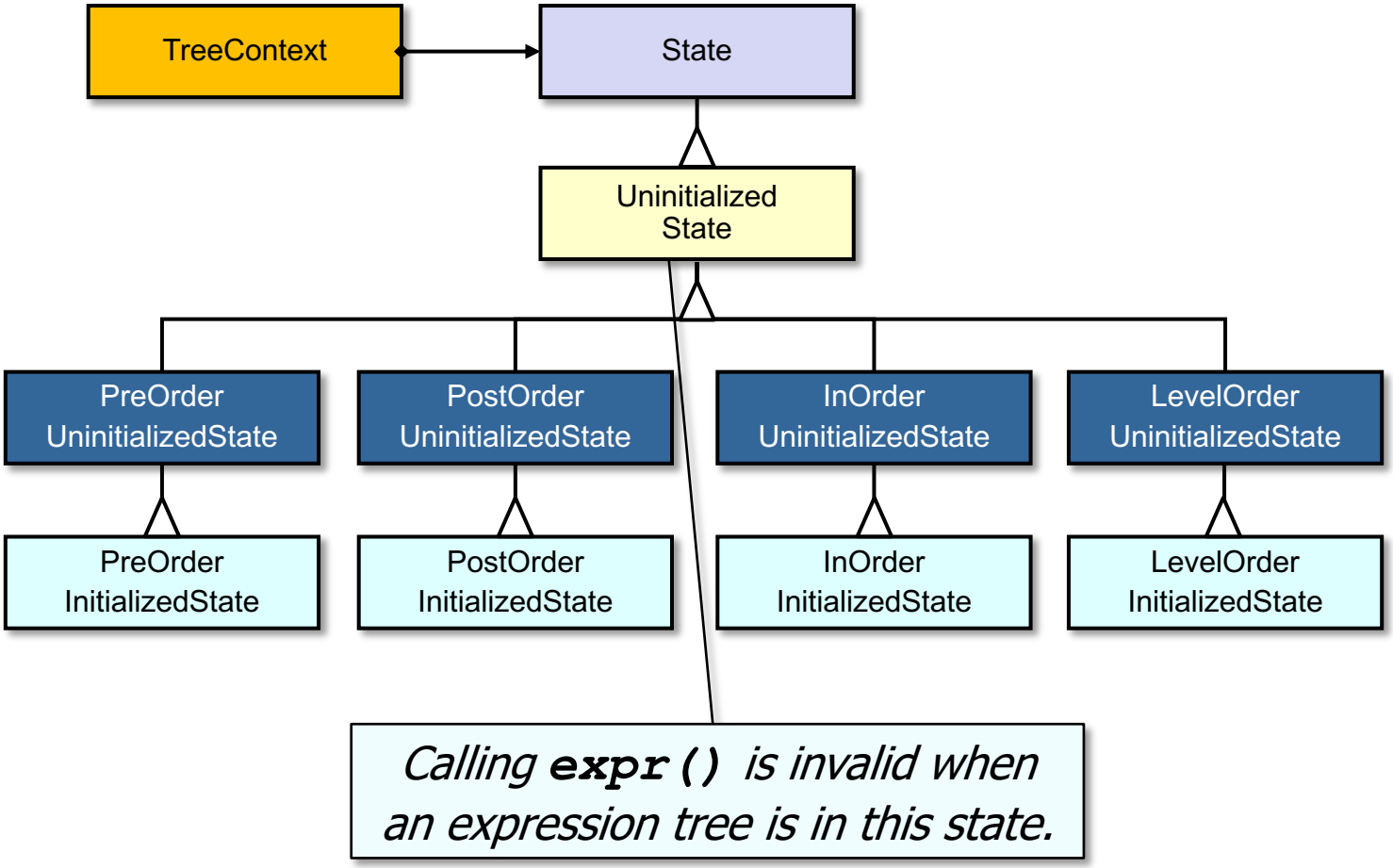
Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *State* pattern can be applied to ensure user requests follow the correct protocol.
- Understand the structure and functionality of the *State* pattern.
- Know how to implement the *State* pattern in Java.
- Be aware of other considerations when applying the *State* pattern.

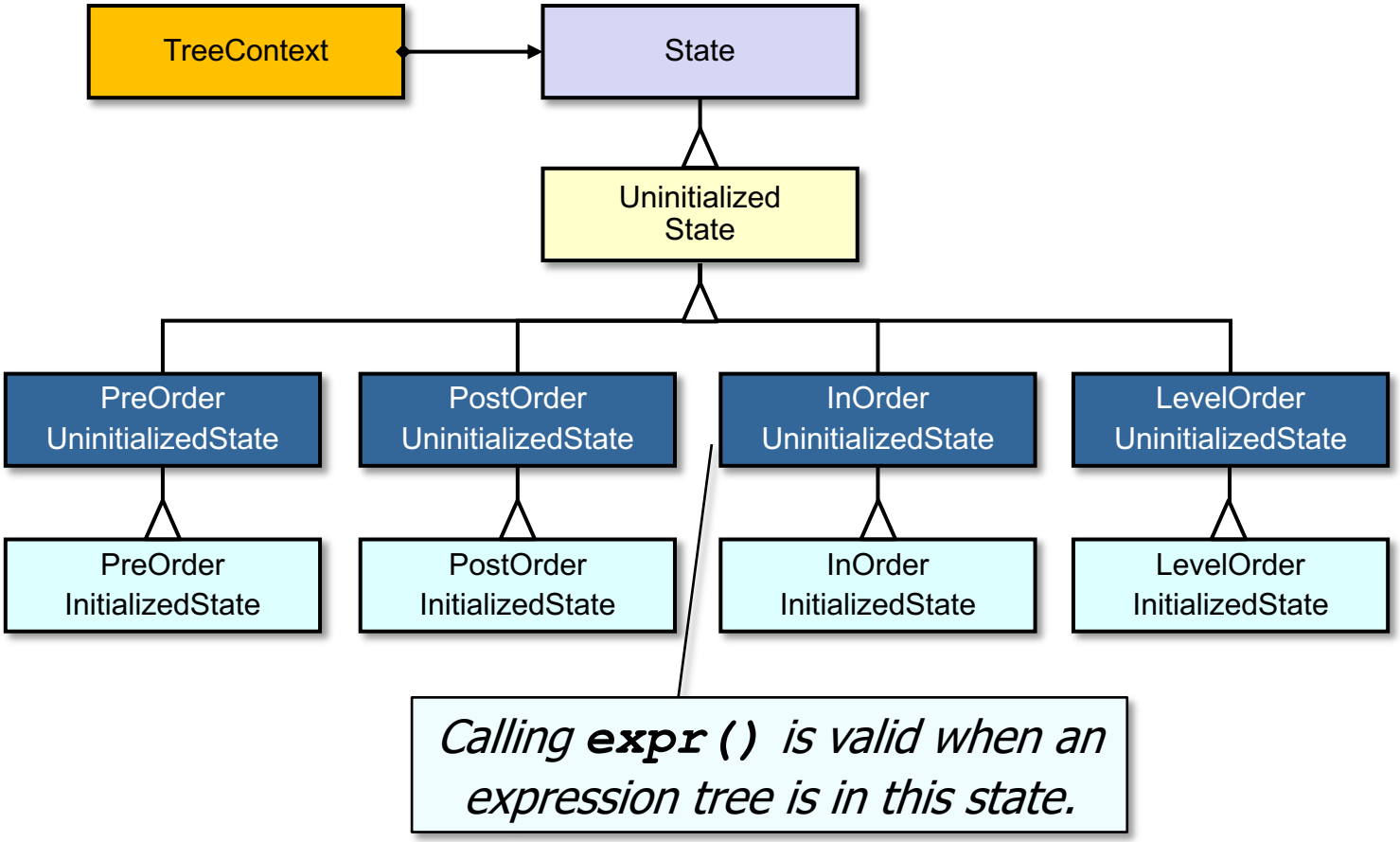
Consequences

+ Localizes state-specific behavior and partitions behavior for different states



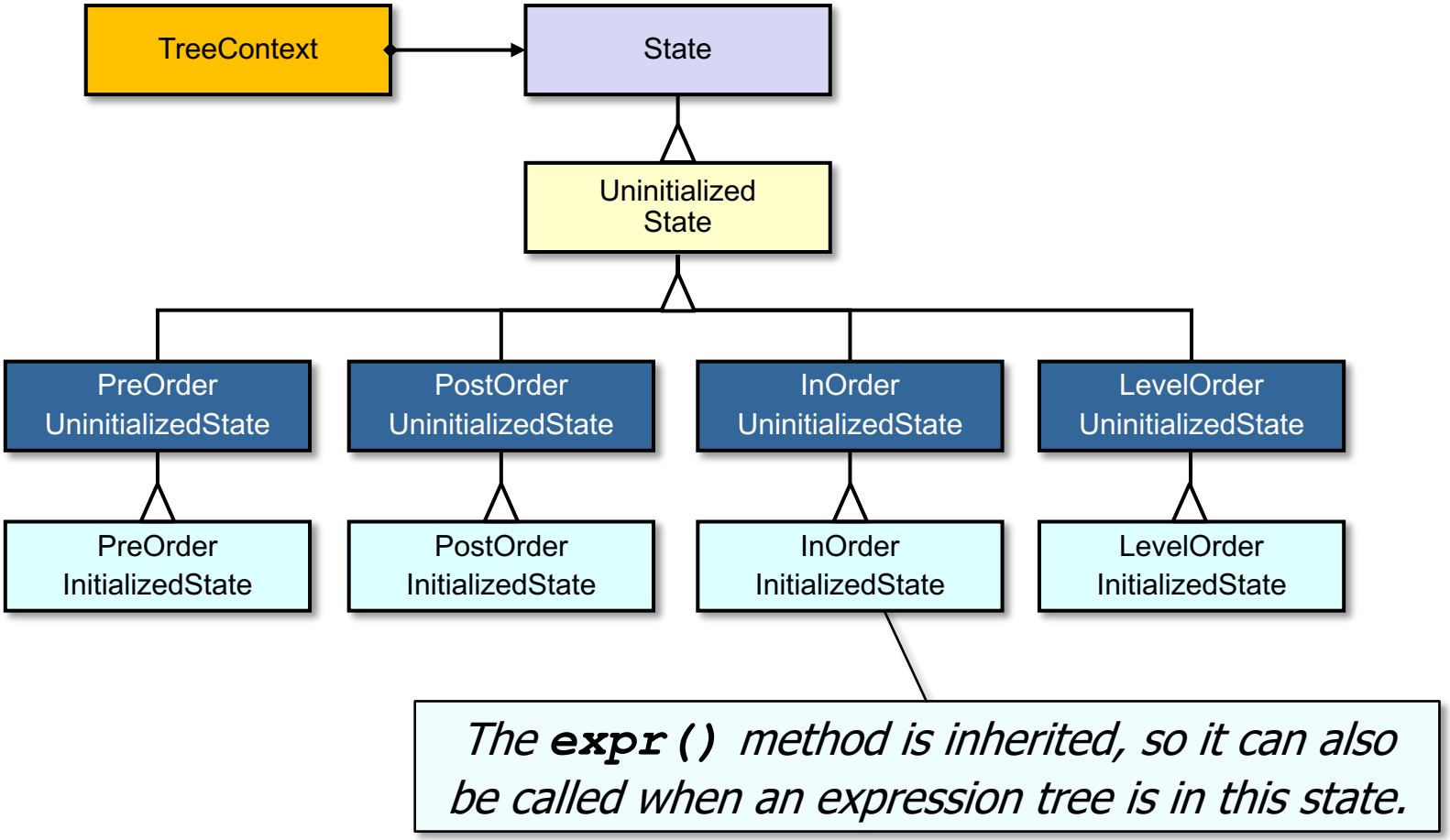
Consequences

+ Localizes state-specific behavior and partitions behavior for different states



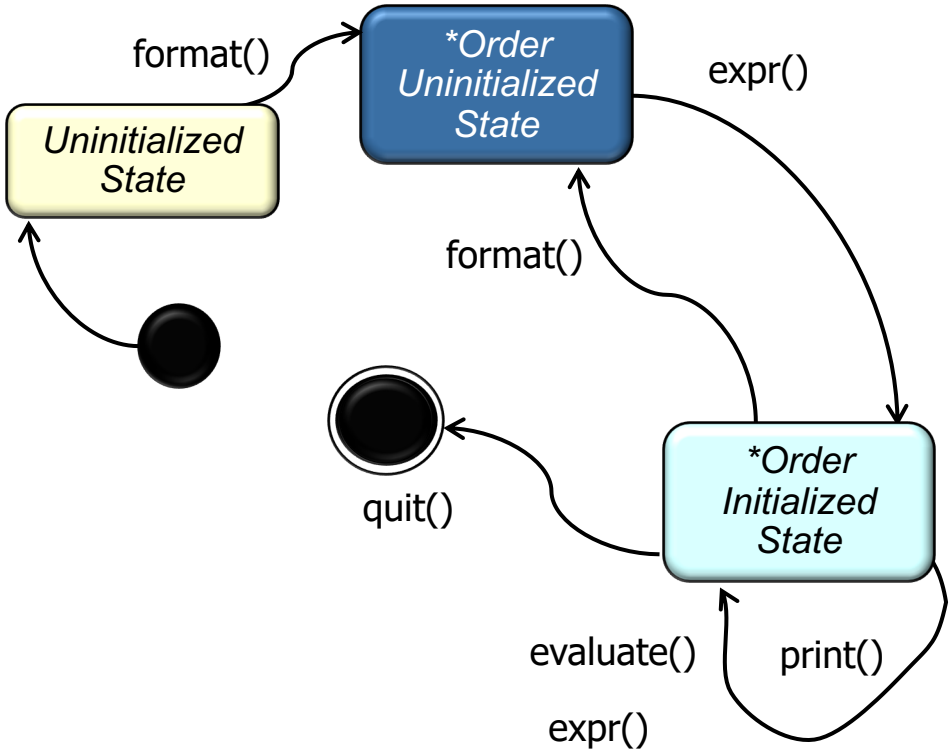
Consequences

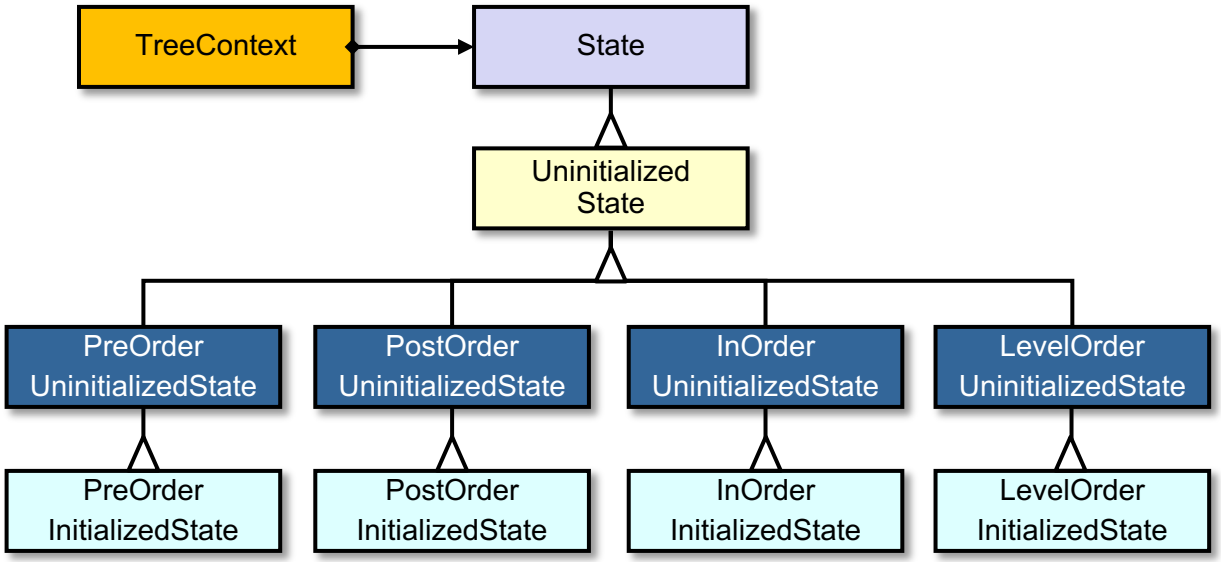
+ State objects and methods can be shared



Consequences

+ Makes state transitions explicit
  - e.g., embodied in the structure of the program's (sub)classes

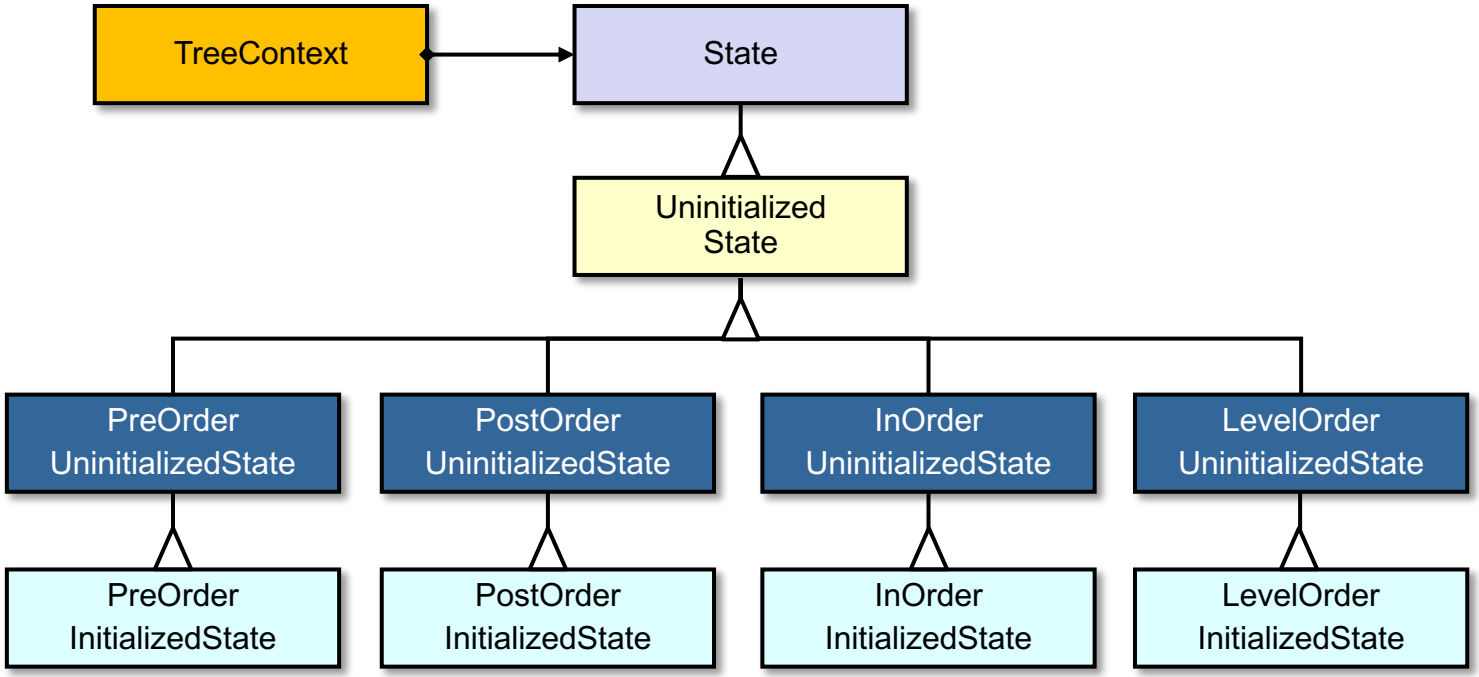




Consequences

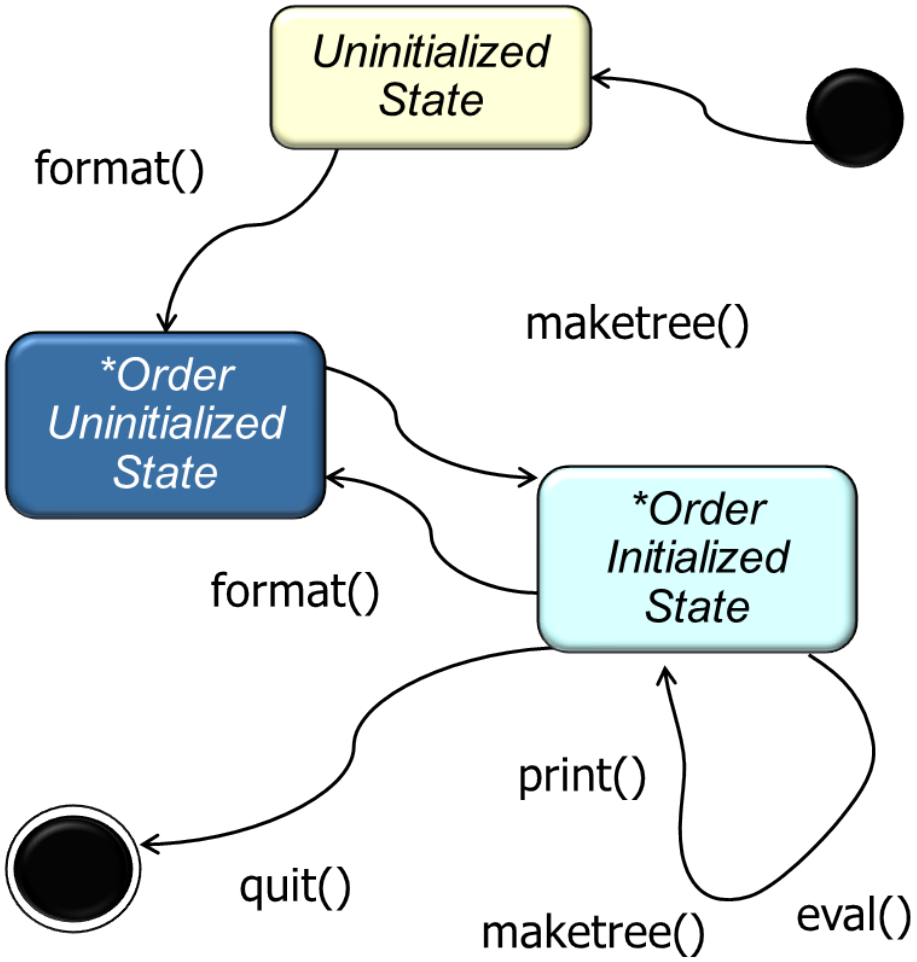
– Can result in many subclasses

- These subclass dependencies & interactions may be hard to understand & evolve if they aren't well-documented (& if the State pattern isn't well-understood)



Implementation considerations

- Where are state transitions defined?



Implementation considerations

- Where are state transitions defined?

```
class InOrderUninitializedState
      extends UninitializedState {
  public void expr
            (TreeContext treeContext,
             String inputExpression) {
    treeContext.tree
      (treeContext
       .interpreter()
       .interpret(inputExpression));

    treeContext.state
      (new InOrderInitializedState());
  }
}
```

e.g., should methods in State subclasses hard-code transitions (as shown here) or should transitions be guided by state tables?

See en.wikipedia.org/wiki/State_transition_table

Implementation considerations

- Creating and destroying state objects
  - Consider using pre-initialized factories

```
class UninitializedStateFactory {
  private HashMap<String, State>
    mUninitStateMap = new HashMap<>();

  UninitializedStateFactory() {
    mUninitStateMap.put("in-order",
       new InOrderUninitializedState());
    ...
  }

  State makeUninitializedState
                      (String format)
  { return mUninitStateMap.get(format);}
  ...
```

```
public class UninitializedState implements State {
  public void format(TreeContext context, String format) {
    context.state(mUninitializedStateFactory
                  .makeUninitializedState(format));
  } ...
```

Implementation considerations

- Creating and destroying state objects
  - Consider using pre-initialized factories

Initialize a hash map that's used by a factory method.

```
class UninitializedStateFactory {
  private HashMap<String, State>
    mUninitStateMap = new HashMap<>();

  UninitializedStateFactory() {
    mUninitStateMap.put("in-order",
       new InOrderUninitializedState());
    ...
  }

  State makeUninitializedState
                        (String format)
  { return mUninitStateMap.get(format);}
  ...
```

```
public class UninitializedState implements State {
  public void format(TreeContext context, String format) {
    context.state(mUninitializedStateFactory
                  .makeUninitializedState(format));
  } ...
```

Implementation considerations

- Creating and destroying state objects
  - Consider using pre-initialized factories

Factory method returns the state associated with a given format.

```
class UninitializedStateFactory {
  private HashMap<String, State>
    mUninitStateMap = new HashMap<>();

  UninitializedStateFactory() {
    mUninitStateMap.put("in-order",
       new InOrderUninitializedState());
    ...
  }

  State makeUninitializedState
                      (String format)
  { return mUninitStateMap.get(format);}
  ...
```

```
public class UninitializedState implements State {
  public void format(TreeContext context, String format) {
    context.state(mUninitializedStateFactory
                  .makeUninitializedState(format));
  } ...
```

Implementation considerations

- Creating and destroying state objects
  - Consider using pre-initialized factories

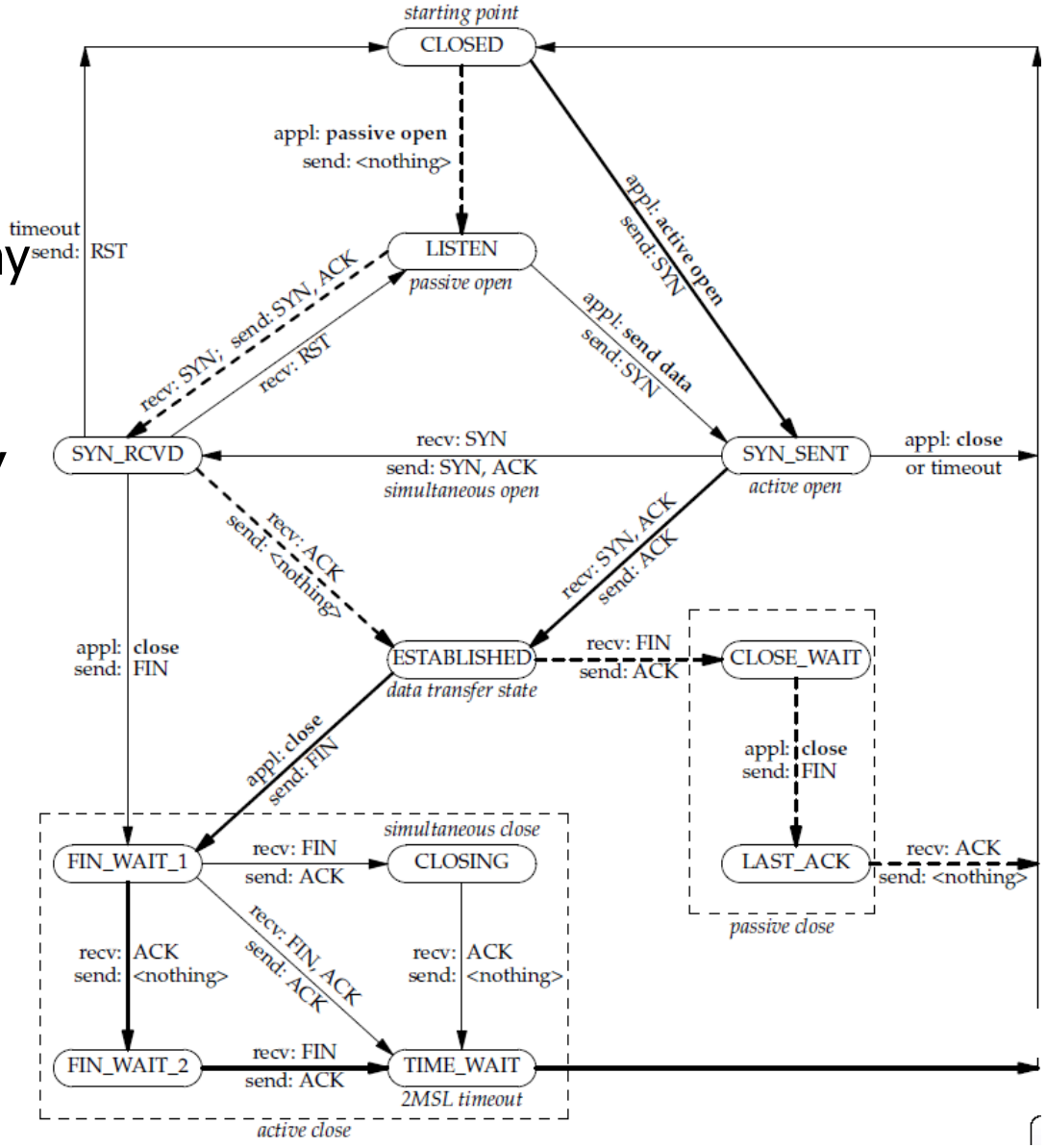
```
class UninitializedStateFactory {
  private HashMap<String, State>
    mUninitStateMap = new HashMap<>();

  UninitializedStateFactory() {
    mUninitStateMap.put("in-order",
       new InOrderUninitializedState());
    ...
  }

  State makeUninitializedState
                      (String format)
  { return mUninitStateMap.get(format);}
  ...
```

```
public class UninitializedState implements State {
  public void format(TreeContext context, String format) {
    context.state(mUninitializedStateFactory
                  .makeUninitializedState(format));
  } ...
```

Use the factory method to set the uninitialized state.

Known uses

- The *State* pattern has been applied to the TCP protocol
  - See Ralph Johnson and Johnny Zweig's article 'Delegation in C++' (*Journal of Object-Oriented Programming*, 4(11), 22–35, November 1991)



See www.dre.vanderbilt.edu/~schmidt/PDF/joop0407.pdf for the article.

Known uses

- The *State* pattern has been applied to the TCP protocol
- Unidraw and Hotdraw drawing tools
- The method javax.faces.lifecycle.LifeCycle#execute()

Class Lifecycle

java.lang.Object
    javax.faces.lifecycle.Lifecycle

Direct Known Subclasses:

`LifecycleWrapper`

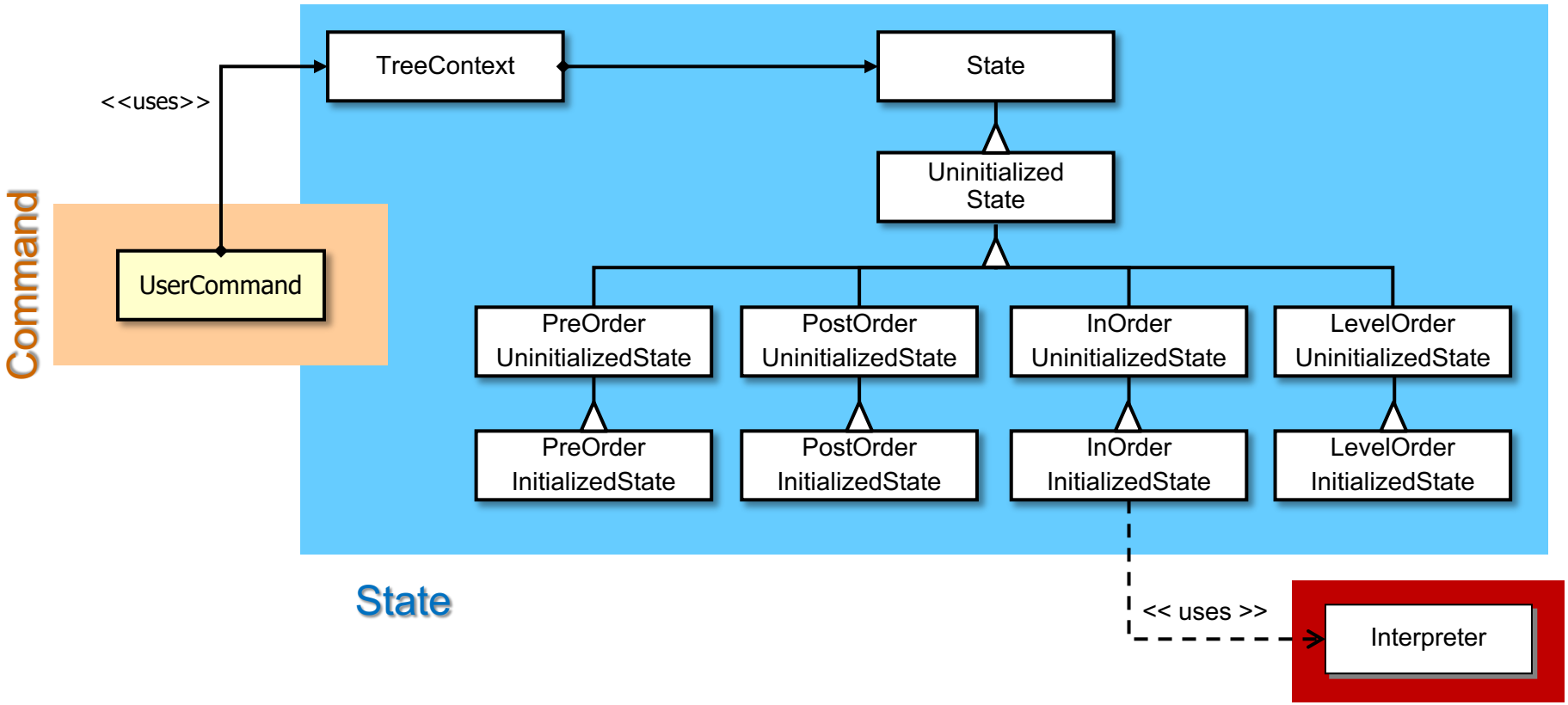
```
public abstract class Lifecycle
extends Object
```

Lifecycle manages the processing of the entire lifecycle of a particular JavaServer Faces request. It is responsible for executing all of the phases that have been defined by the JavaServer Faces Specification, in the specified order, unless otherwise directed by activities that occurred during the execution of each phase.

An instance of `Lifecycle` is created by calling the `getLifecycle()` method of `LifecycleFactory`, for a specified lifecycle identifier. Because this instance is shared across multiple simultaneous requests, it must be implemented in a thread-safe manner.

Summary of State Pattern

State ensures that user requests are performed in the correct order.



State structures valid sequencing of user requests into class hierarchy design.

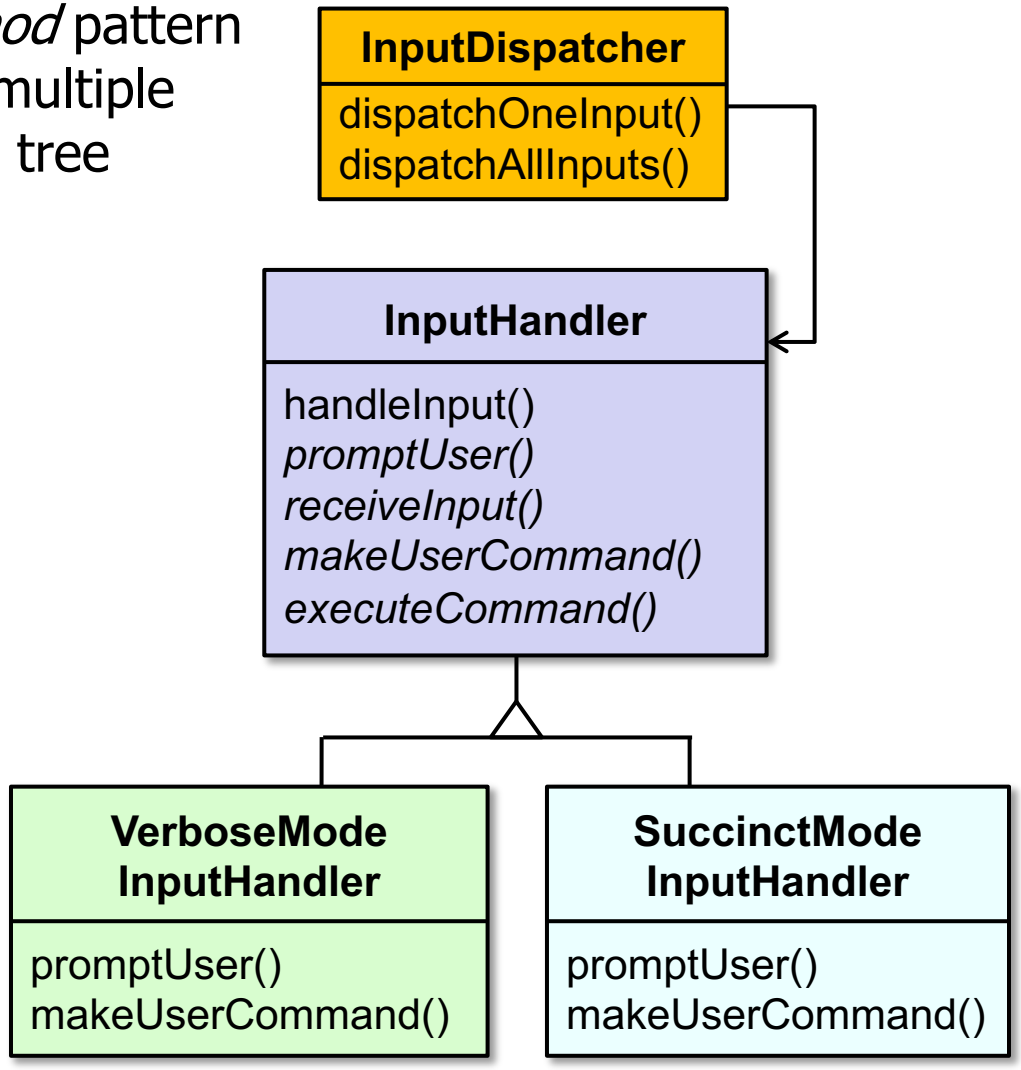
The Template Method Pattern

Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Template Method* pattern can be applied to flexibly support multiple operating modes in the expression tree processing app.

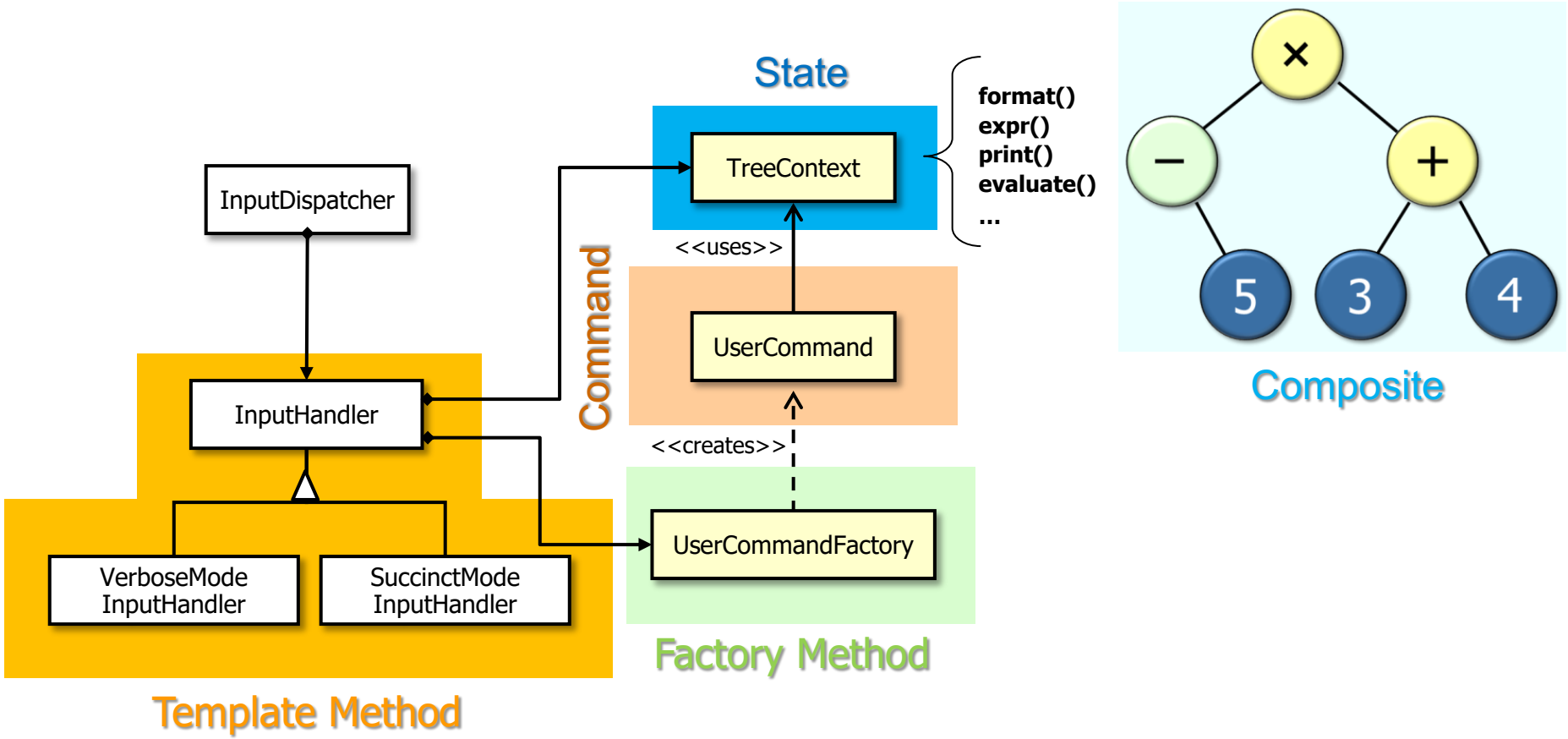


Douglas C. Schmidt

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

A Pattern for Encapsulating Algorithm Variability

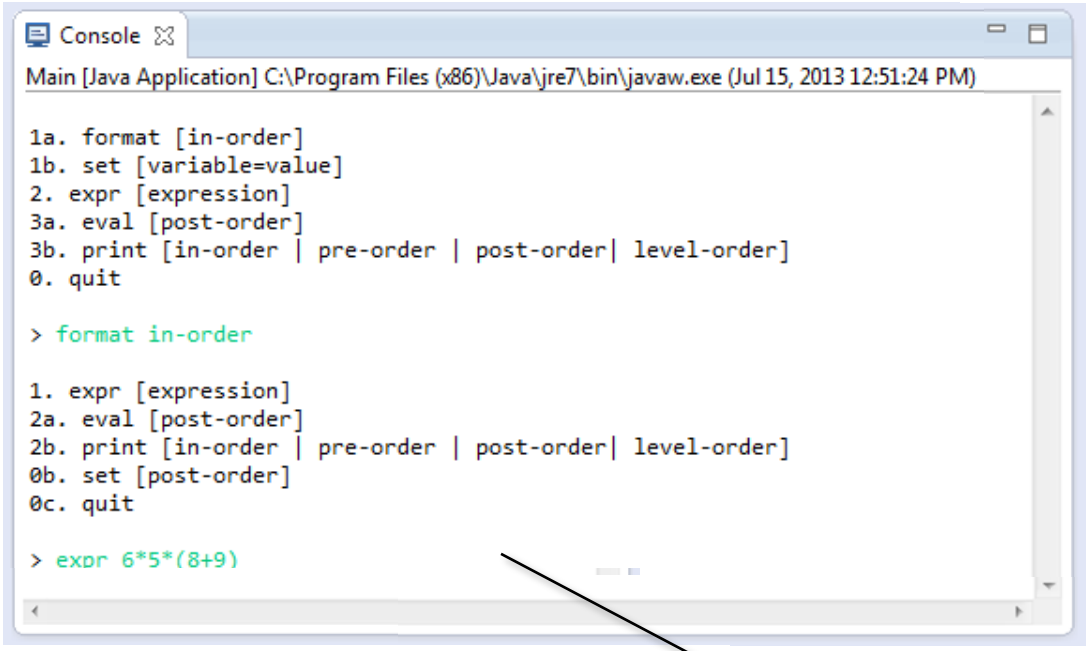
***Purpose**: Factor out common code to support multiple operating modes (succinct vs. verbose).*



Template Method supports controlled variability of steps in an algorithm.

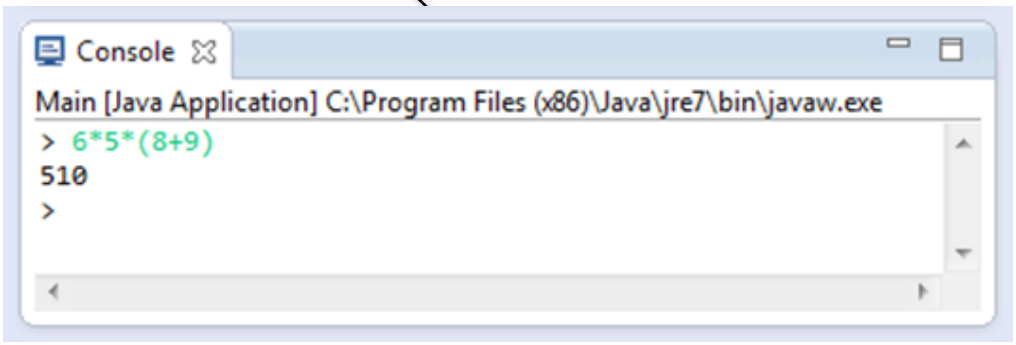
Context: OO Expression Tree Processing App

- This app has two primary operating modes: *verbose* and *succinct*.



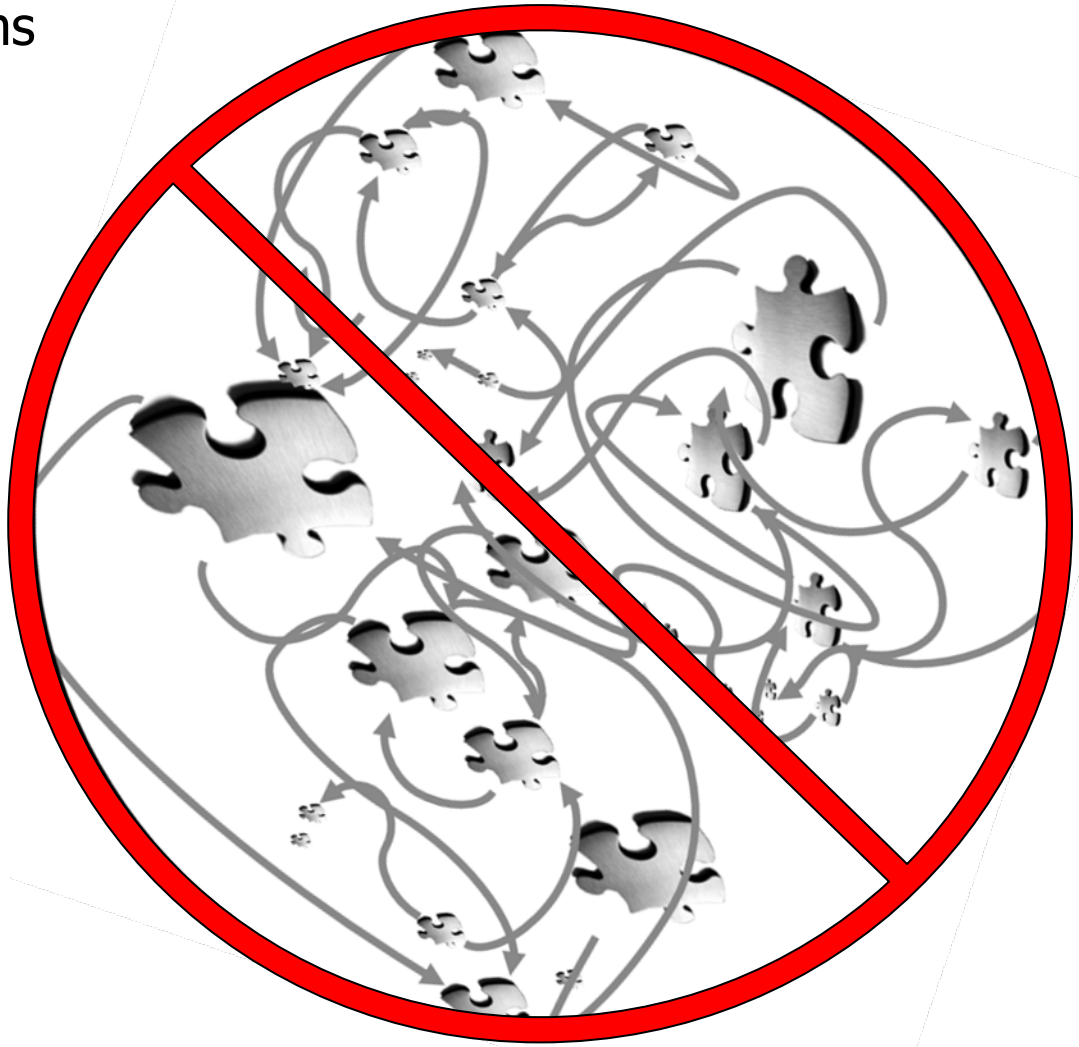
Verbose mode

Succinct mode



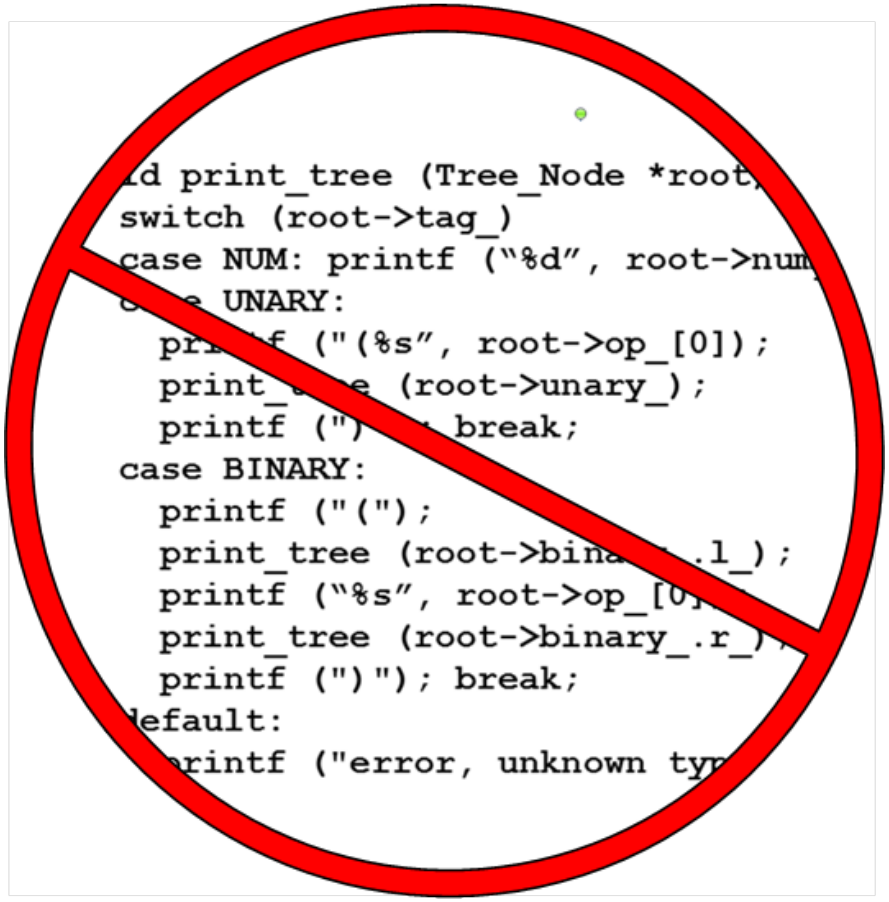
Problem: Non-Extensible Operating Modes

- Structuring the program in terms of the two operating modes' algorithms is problematic.



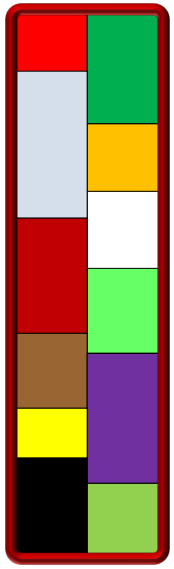
Problem: Non-Extensible Operating Modes

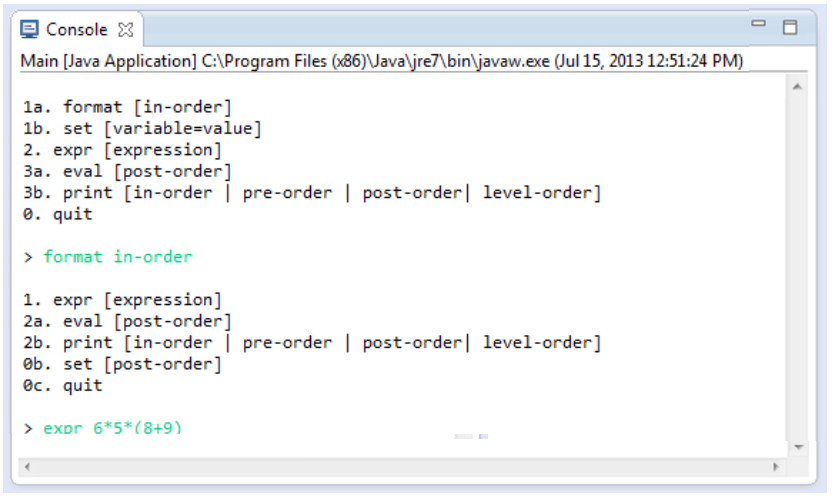
- Structuring the program in terms of the two operating modes' algorithms is problematic, e.g.,
  - Incurs many of the same limitations as algorithmic decomposition
    - e.g., complexity will reside in (variable) algorithms rather than (stable) structure



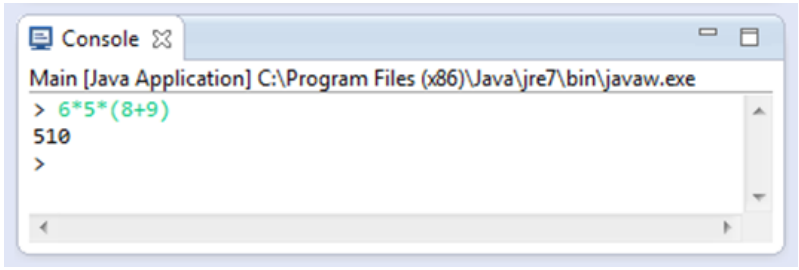
Problem: Non-Extensible Operating Modes

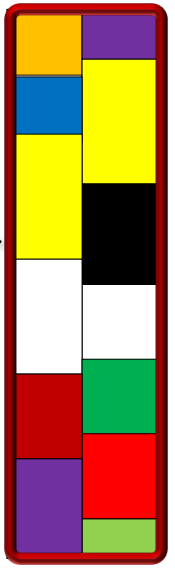
- Structuring the program in terms of the two operating modes' algorithms is problematic, e.g.,
  - Incurs many of the same limitations as algorithmic decomposition
  - Impedes maintainability and evolution of the code base due to "silo'ing"





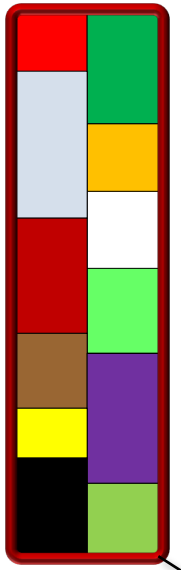
Silo'ing (non-reusable) code

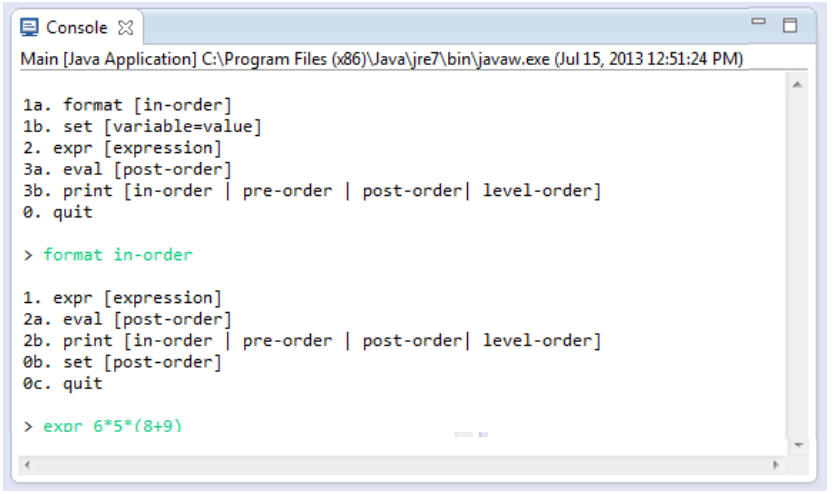




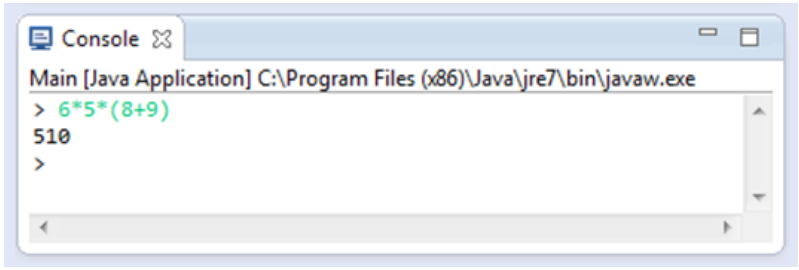
Problem: Non-Extensible Operating Modes

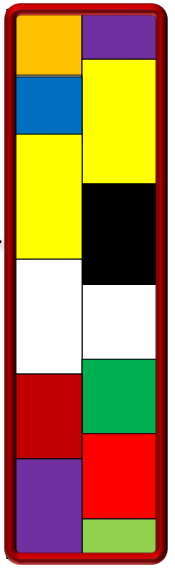
- Structuring the program in terms of the two operating modes' algorithms is problematic, e.g.,
  - Incurs many of the same limitations as algorithmic decomposition
  - Impedes maintainability and evolution of the code base due to "silo'ing", e.g.,
    - Verbose mode algorithm bug fixes and improvements won't be reused by succinct mode algorithms





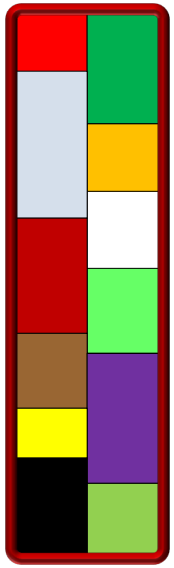
Stovepiped (nonreusable) code

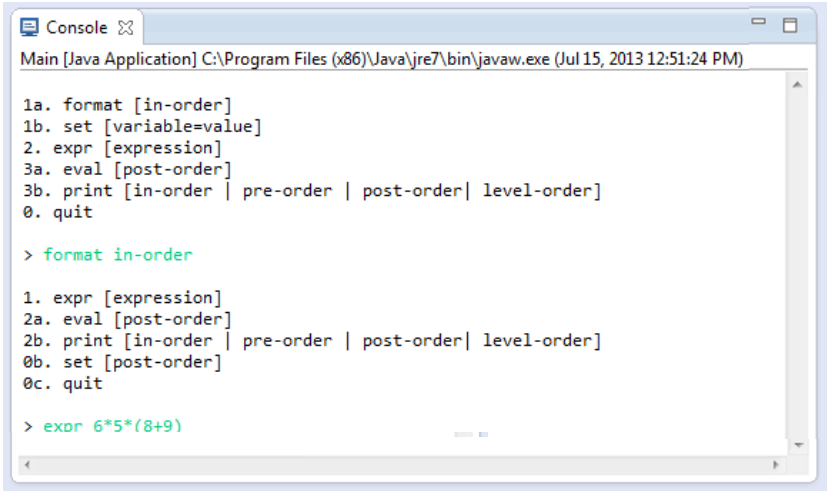


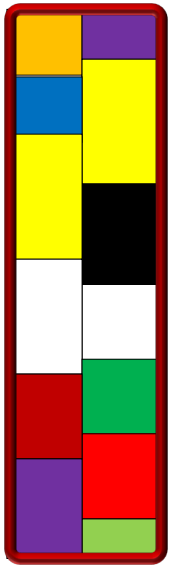


Problem: Non-Extensible Operating Modes

- Structuring the program in terms of the two operating modes' algorithms is problematic, e.g.,
  - Incurs many of the same limitations as algorithmic decomposition
  - Impedes maintainability and evolution of the code base due to "silo'ing", e.g.,
    - Verbose mode algorithm bug fixes and improvements won't be reused by succinct mode algorithms
    - Violates the "Don't Repeat Yourself" (DRY) principle



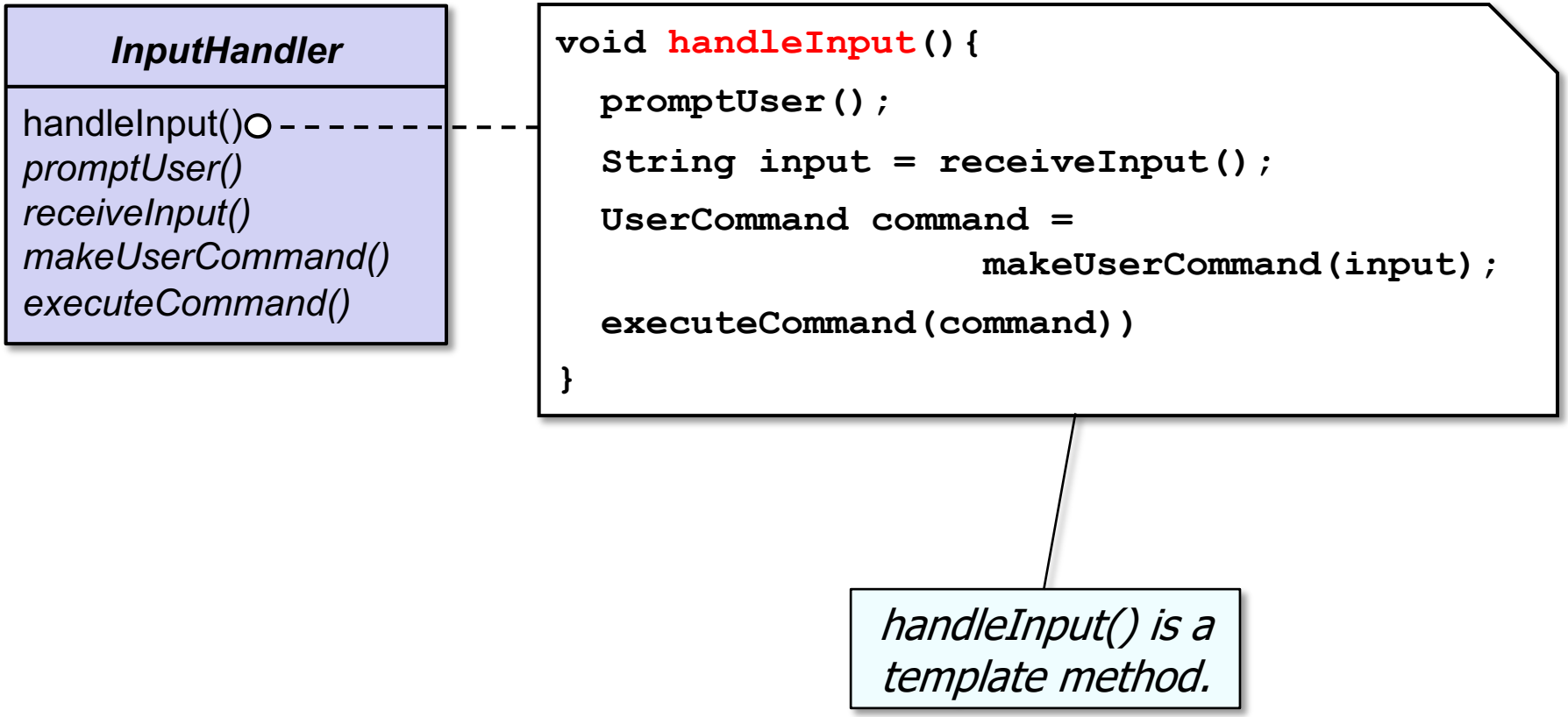




See en.wikipedia.org/wiki/Don't_repeat_yourself

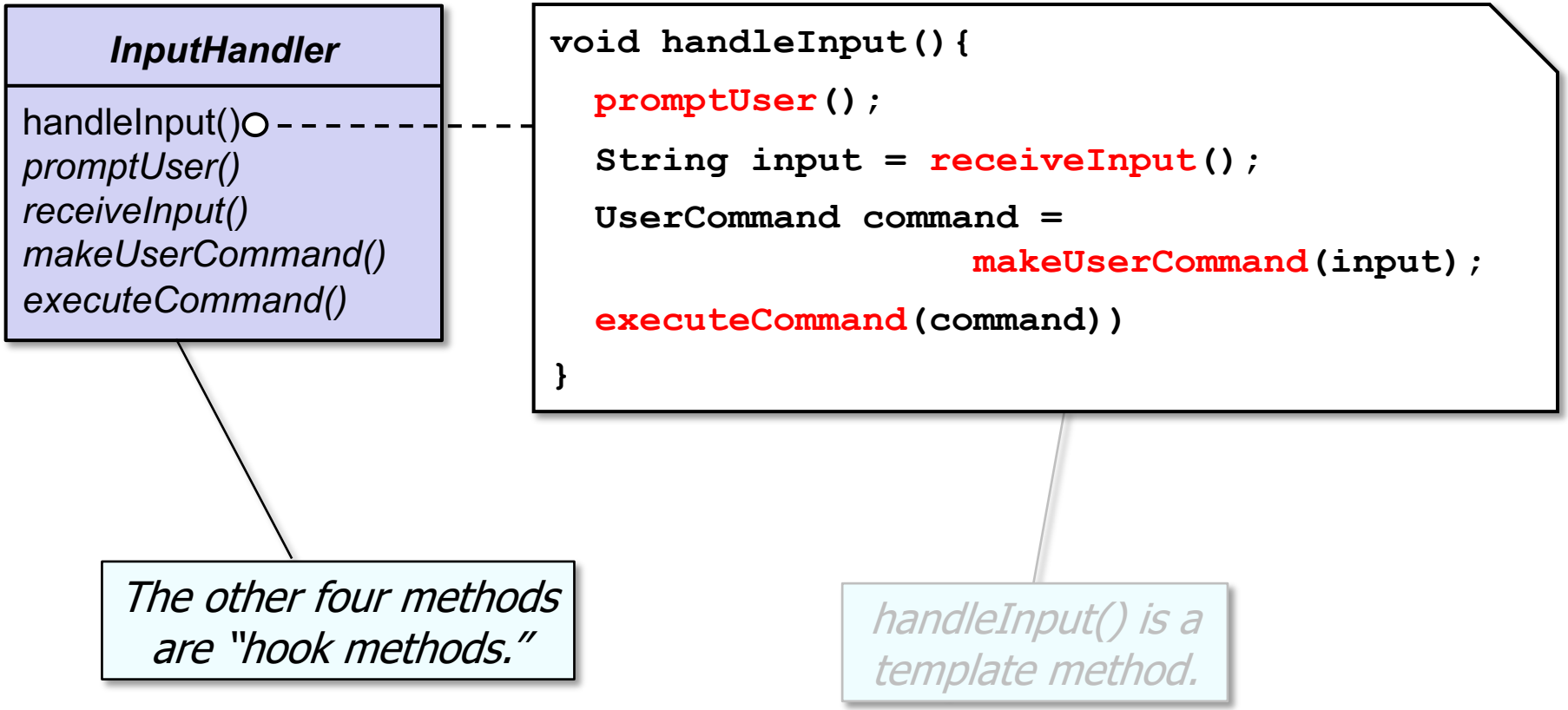
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class



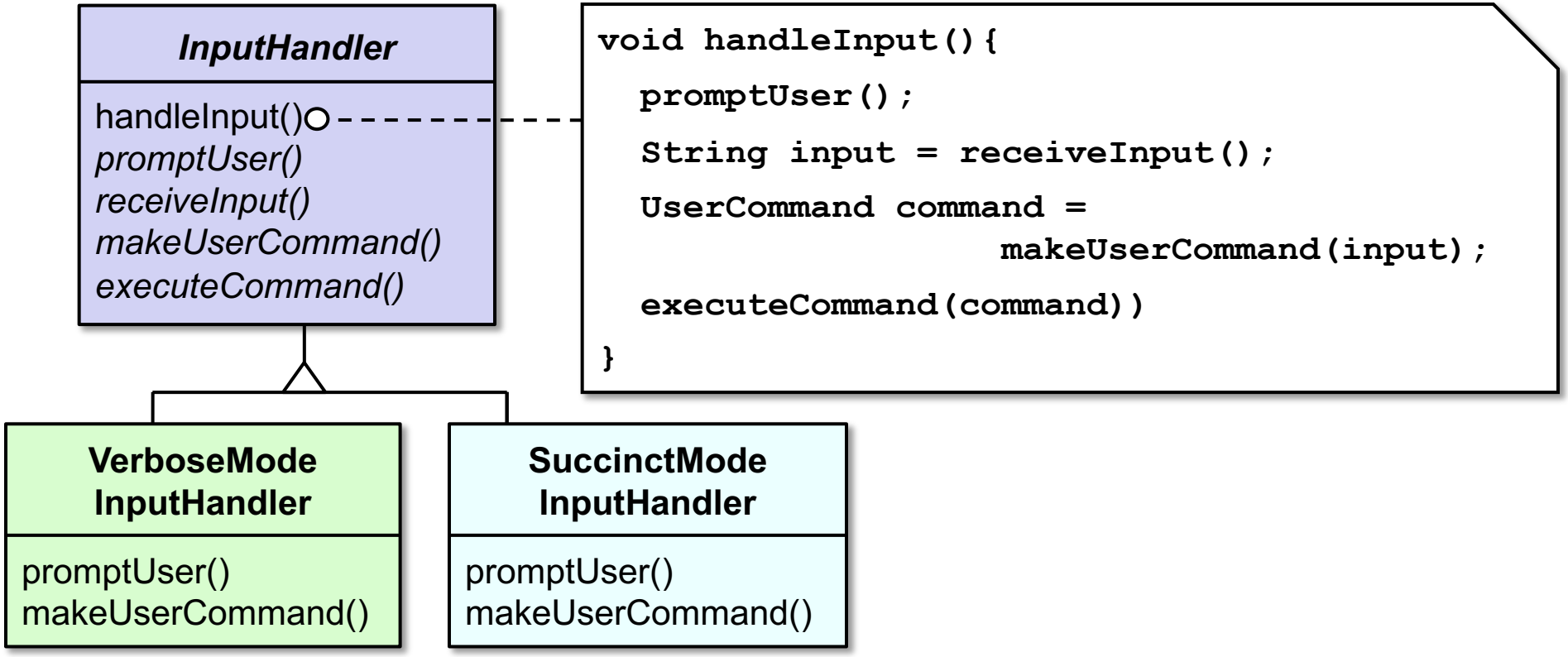
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class and let subclasses define variants.



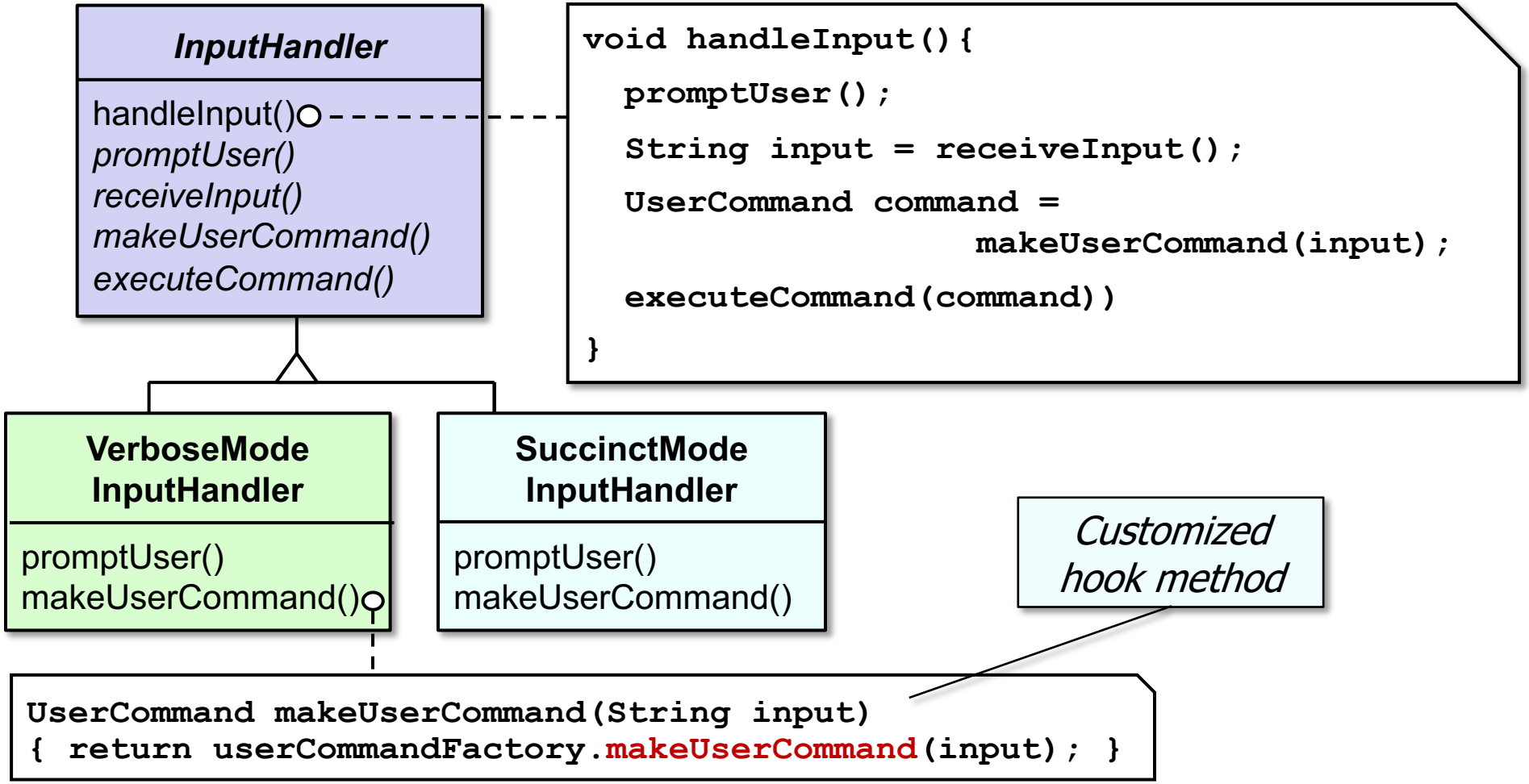
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class and let subclasses define variants.



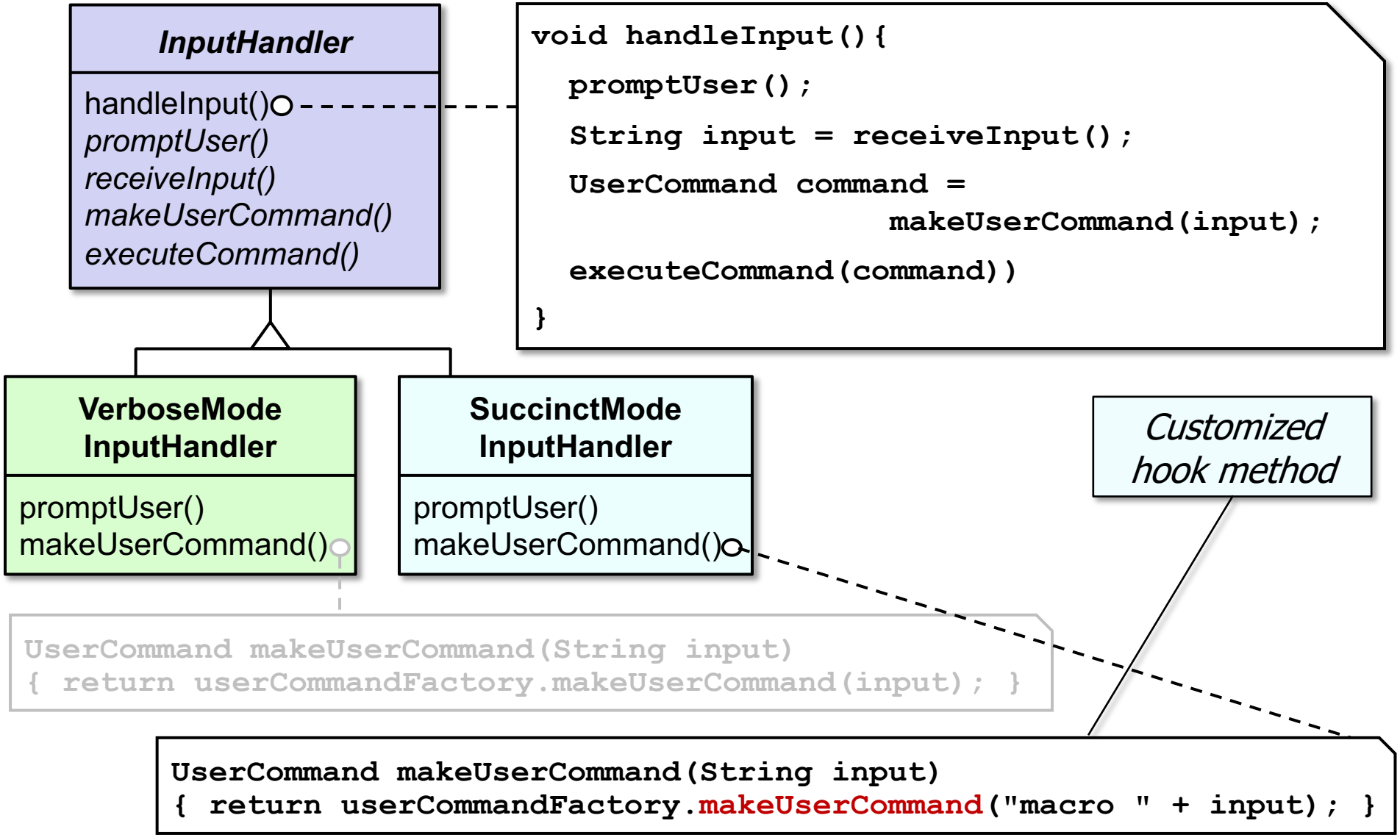
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class and let subclasses define variants.



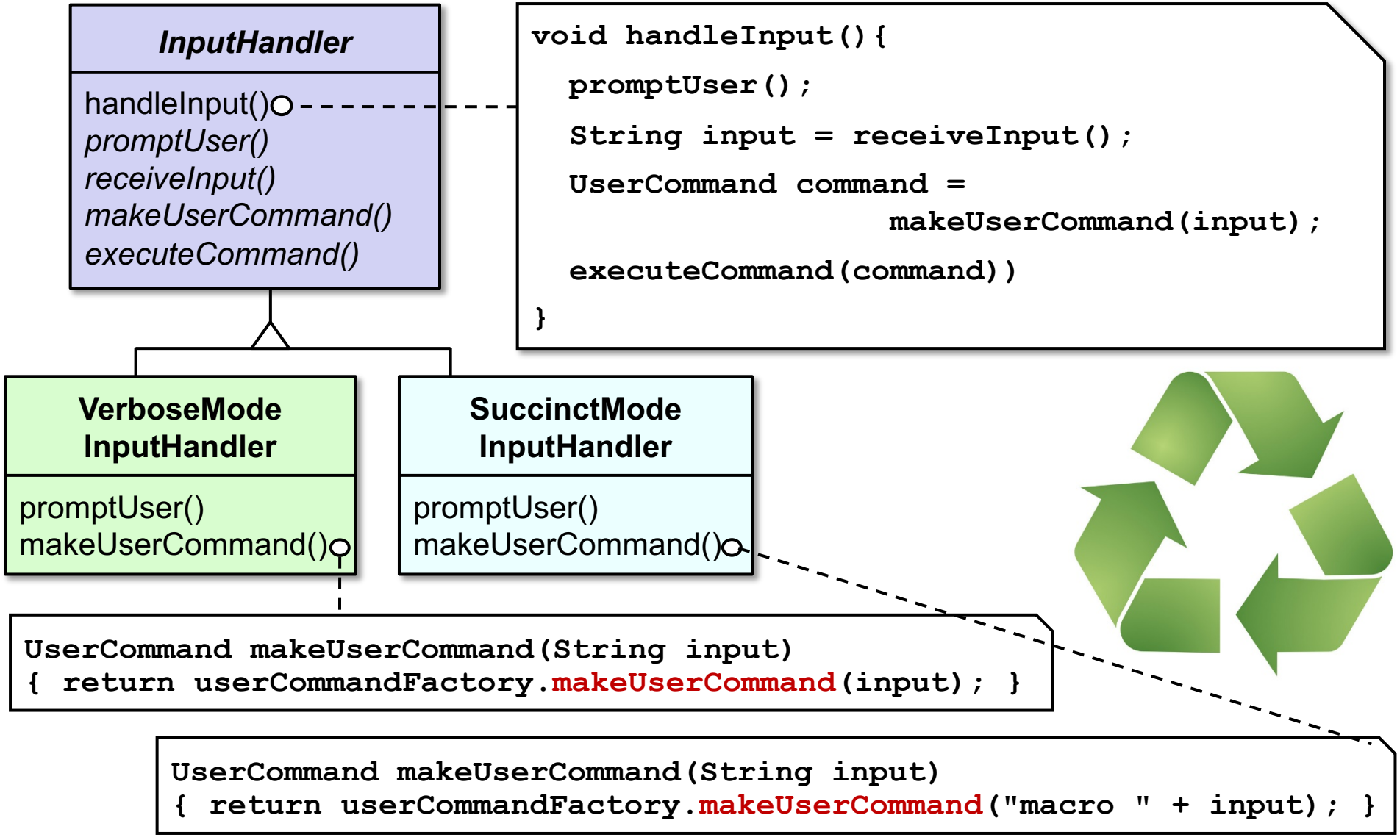
Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class and let subclasses define variants.



Solution: Encapsulate Algorithm Variability

- Implement algorithm once in super class and let subclasses define variants.



This solution increases opportunities for systematic software reuse.

InputHandler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

Class methods

```
                    void handleInput()
                    void promptUser()
                  String receiveInput()
            UserCommand makeUserCommand(String input)
                    void executeCommand(UserCommand command)
  static InputHandler makeHandler(boolean verbose)
```

InputHandler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

Class methods

Template method

```
                 void handleInput()
                 void promptUser()
               String receiveInput()
           UserCommand makeUserCommand(String input)
                 void executeCommand(UserCommand command)
  static InputHandler makeHandler(boolean verbose)
```

InputHandler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

Class methods

```
                  void handleInput()
                  void promptUser()
                String receiveInput()
          UserCommand makeUserCommand(String input)
                  void executeCommand(UserCommand command)
static InputHandler makeHandler(boolean verbose)
```

Hook methods

InputHandler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

Class methods

```
                  void handleInput()
                  void promptUser()
                String receiveInput()
            UserCommand makeUserCommand(String input)
                  void executeCommand(UserCommand command)
  static InputHandler makeHandler(boolean verbose)
```

Factory method

InputHandler Class Overview

- An abstract class that provides the boilerplate algorithm for controlling the operating modes of the expression tree processing app

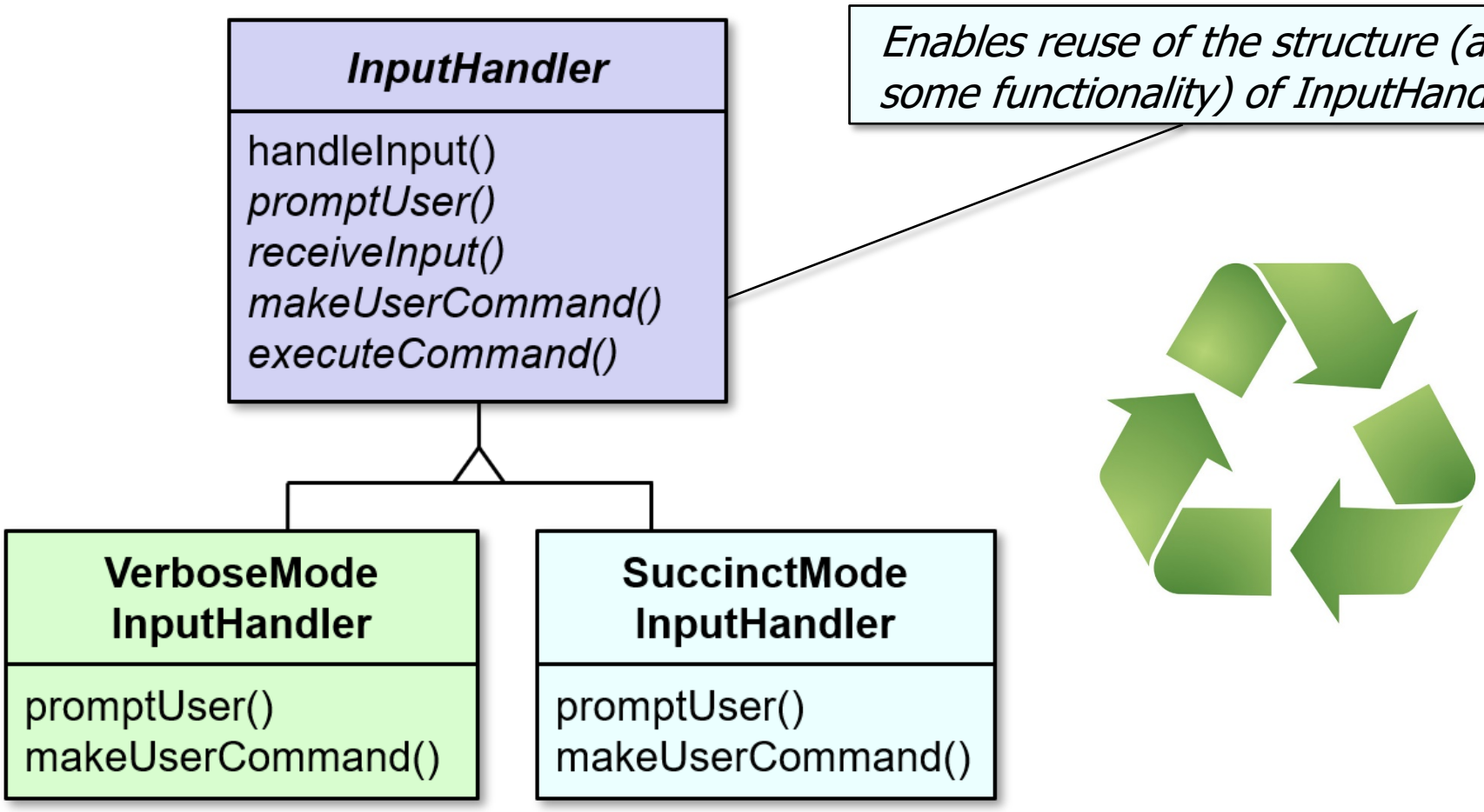
Class methods

```
                    void handleInput()
                    void promptUser()
                  String receiveInput()
            UserCommand makeUserCommand(String input)
                    void executeCommand(UserCommand command)
 static InputHandler makeHandler(boolean verbose)
```

- **Commonality**: provides a common interface for handling user input events and performing steps in the expression tree processing algorithm
- **Variability**: subclasses implement various operating modes, e.g., verbose vs. succinct mode

InputHandler Class Hierarchy Overview

- The subclasses of **`InputHandler`** override several of its hook methods to implement the "verbose" and "succinct" operating modes.



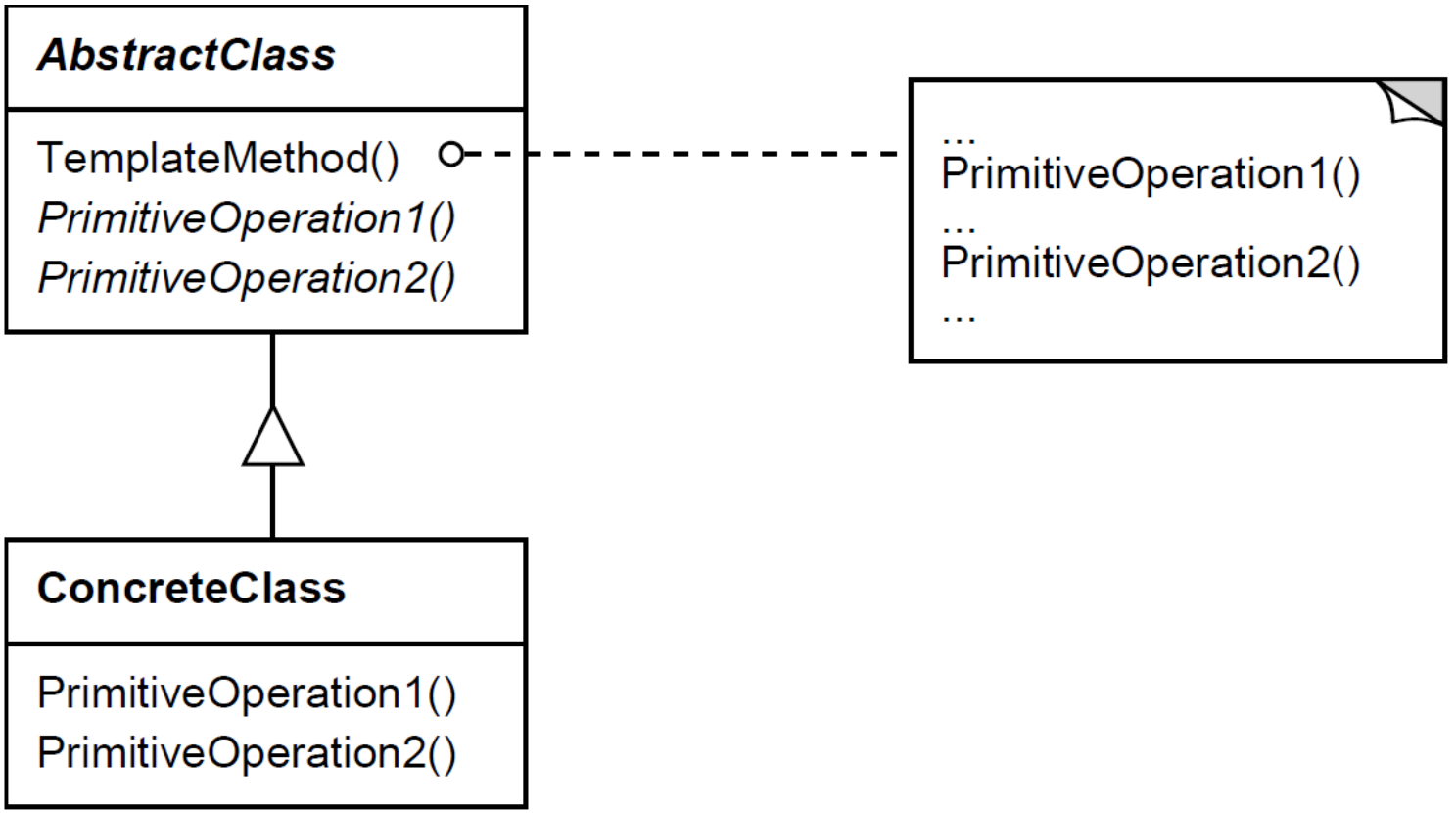
The Template Method Pattern

Structure and Functionality

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Template Method* pattern can be applied to flexibly support multiple operating modes in the expression tree processing app.
- Understand the structure and functionality of the *Template Method* pattern.



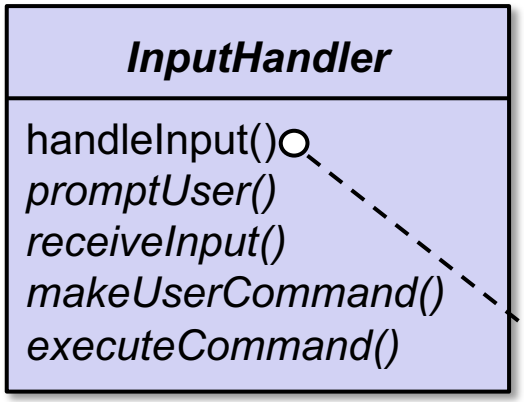
Douglas C. Schmidt

Structure and Functionality of the Template Method Pattern

Template Method GoF Class Behavioral

Intent

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

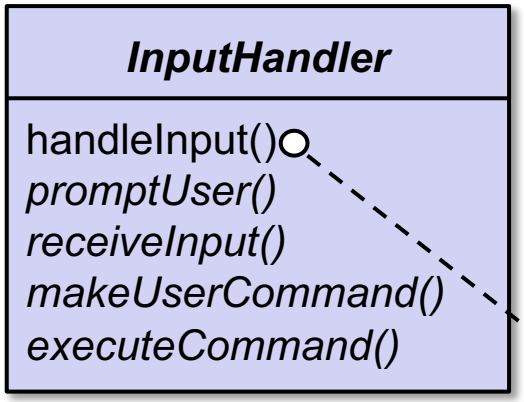



```
void handleInput(){
  promptUser();
  String input = receiveInput();
  UserCommand command =
                    makeUserCommand(input);
  executeCommand(command))
}
```

See en.wikipedia.org/wiki/Template_method_pattern

Applicability

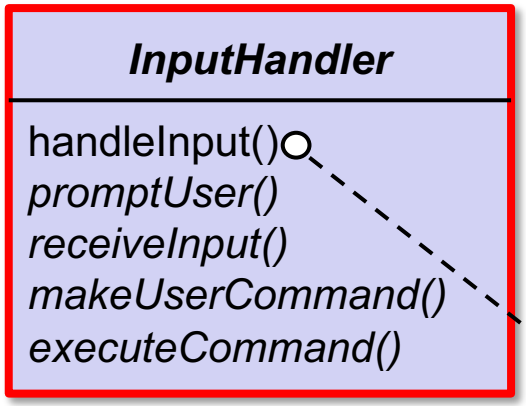
- Implement invariant aspects of an algorithm *once* and let subclasses define variant parts




```
void handleInput(){
  promptUser();
  String input = receiveInput();
  UserCommand command =
                    makeUserCommand(input);
  executeCommand(command))
}
```

Applicability

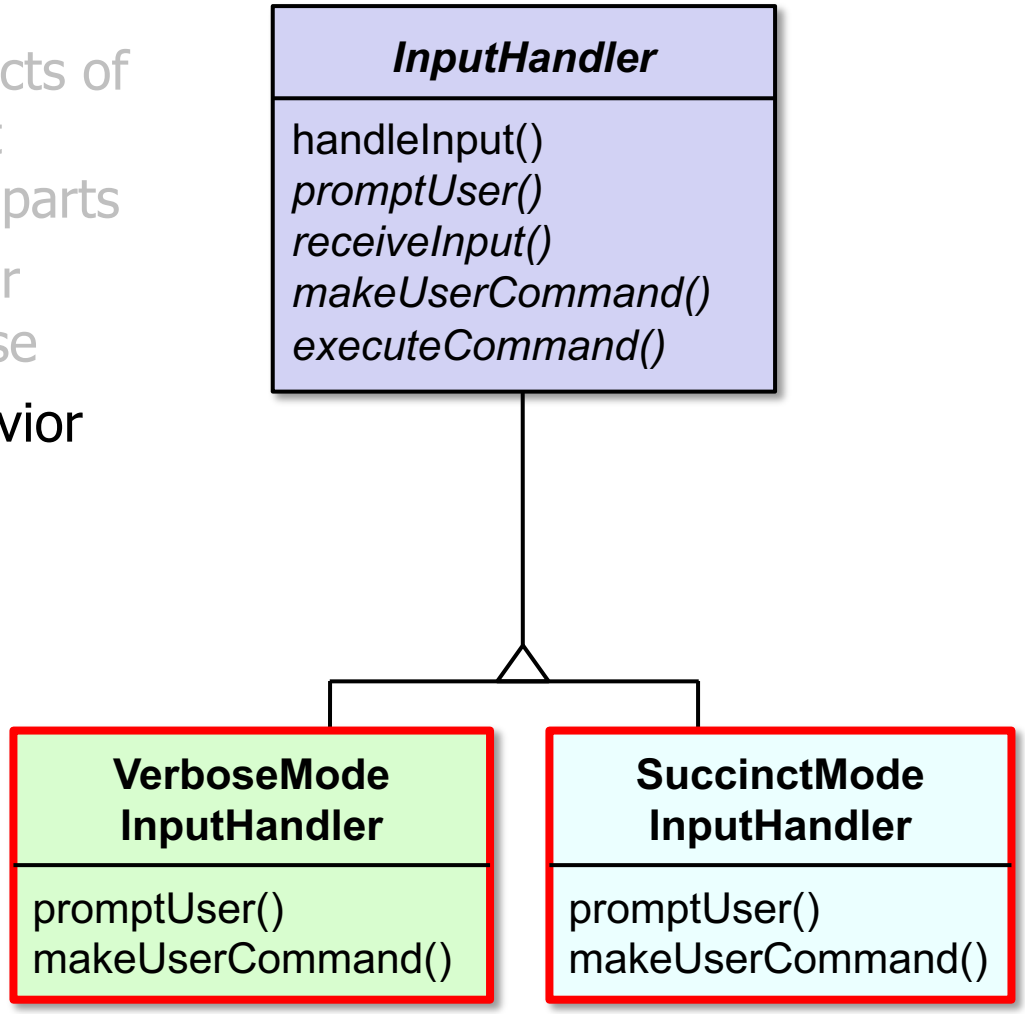
- Implement invariant aspects of an algorithm *once* and let subclasses define variant parts
- Localize common behavior in a class to enhance reuse



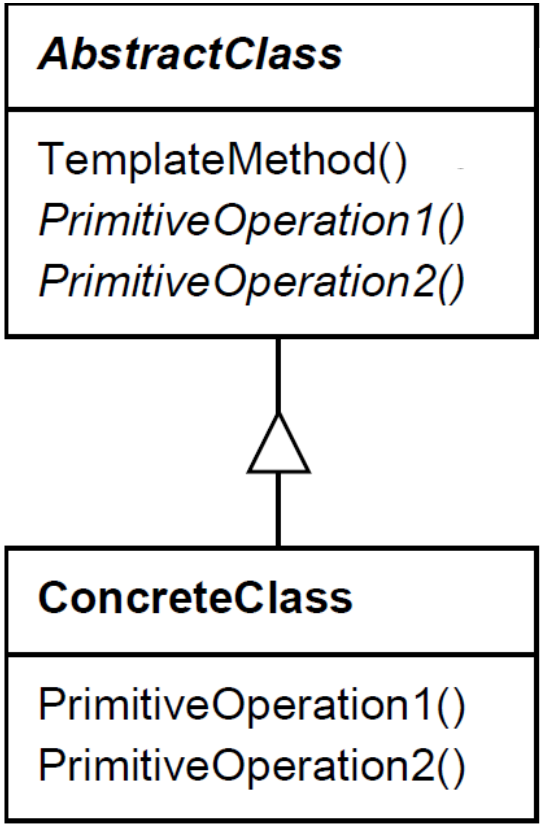

```
void handleInput(){
  promptUser();
  String input = receiveInput();
  UserCommand command =
                    makeUserCommand(input);
  executeCommand(command))
}
```

Applicability

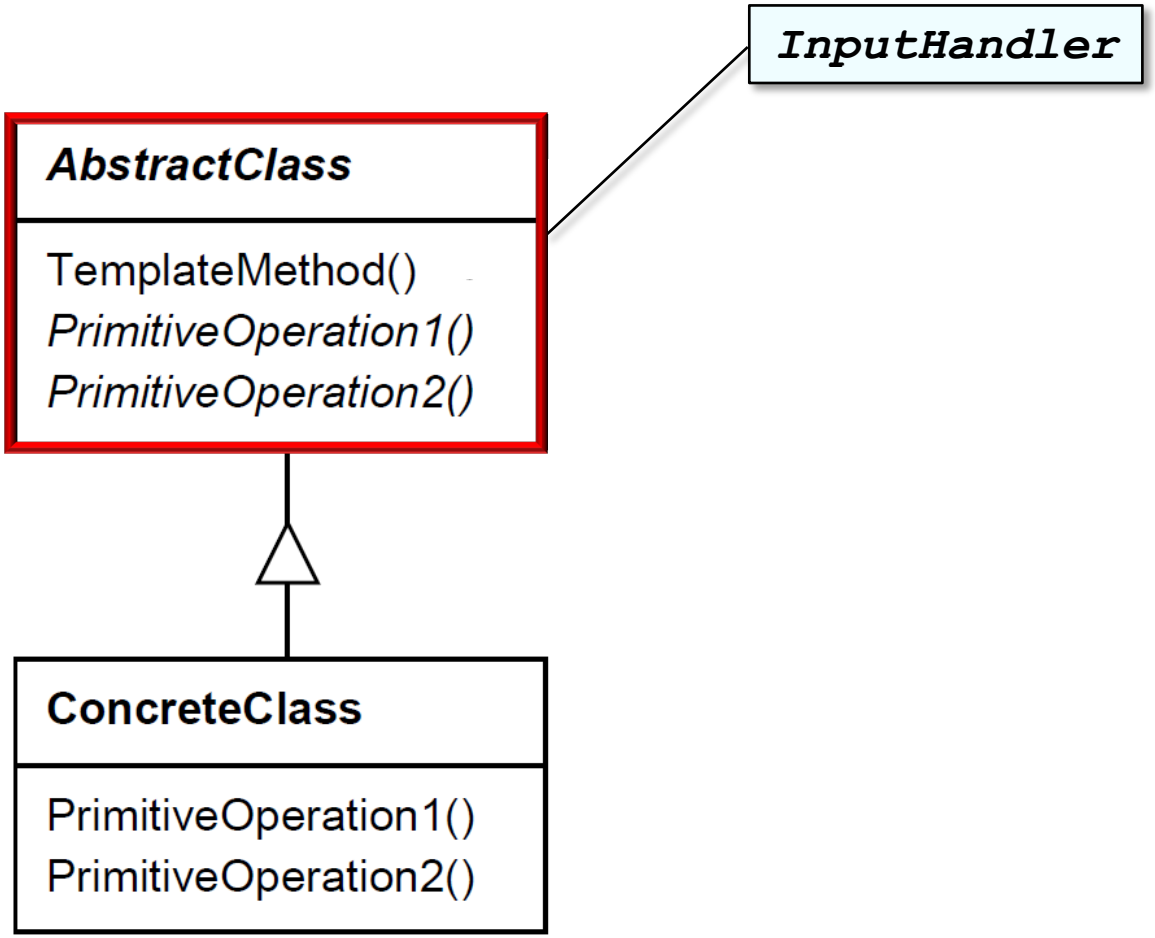
- Implement invariant aspects of an algorithm *once* and let subclasses define variant parts
- Localize common behavior in a class to enhance reuse
- Handle variations in behavior via subclassing



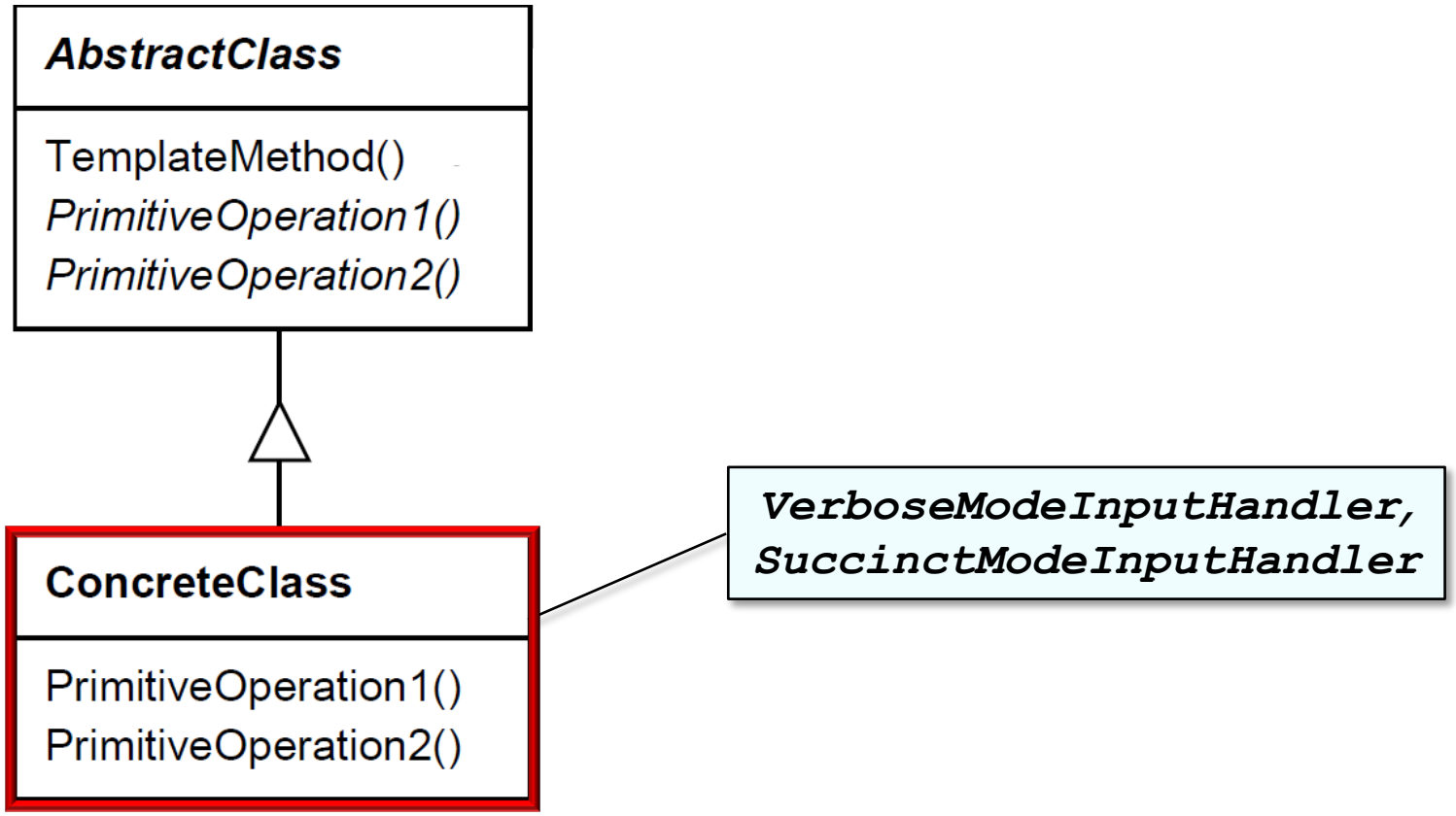
Structure and participants



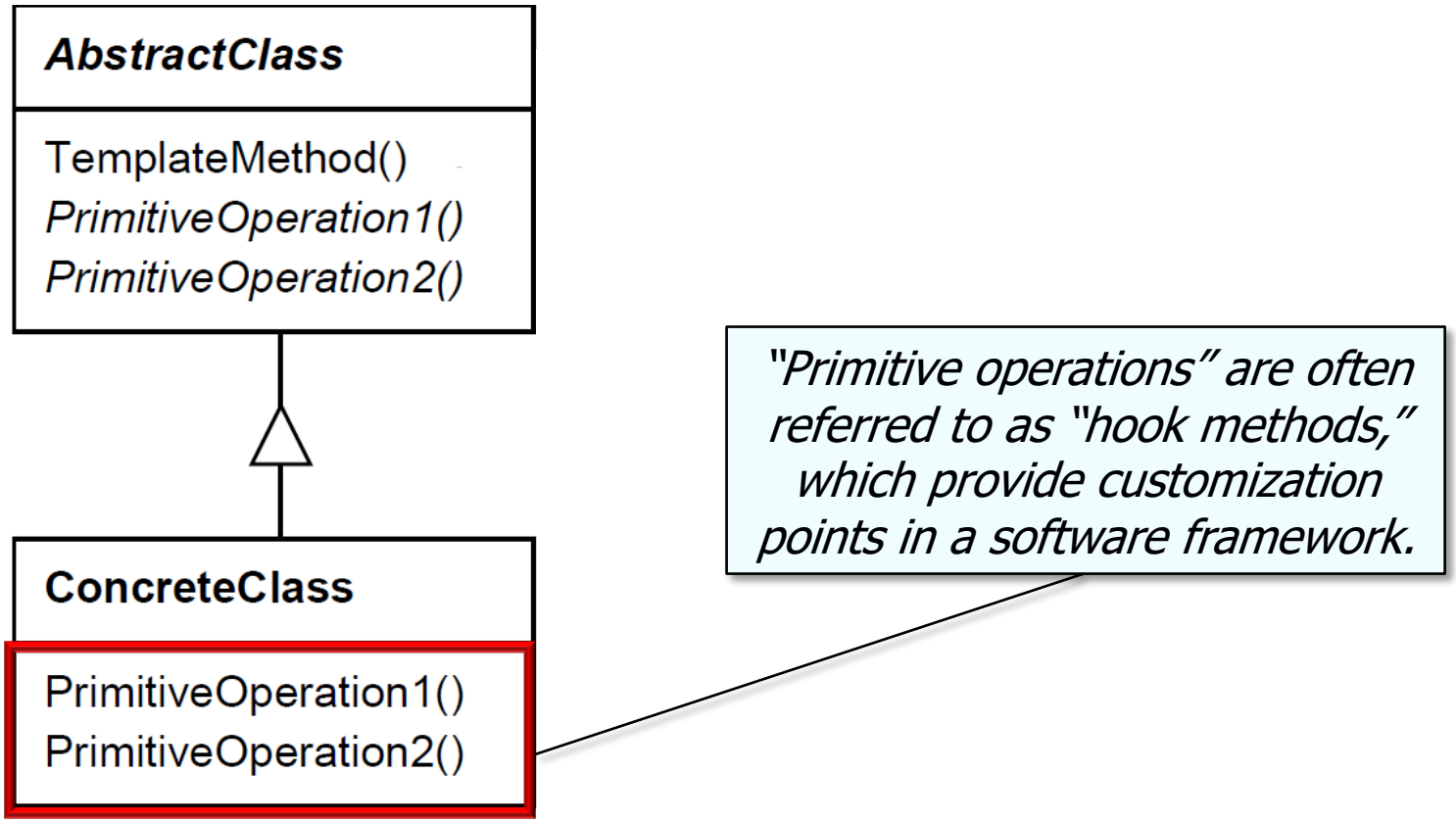
Structure and participants



Structure and participants

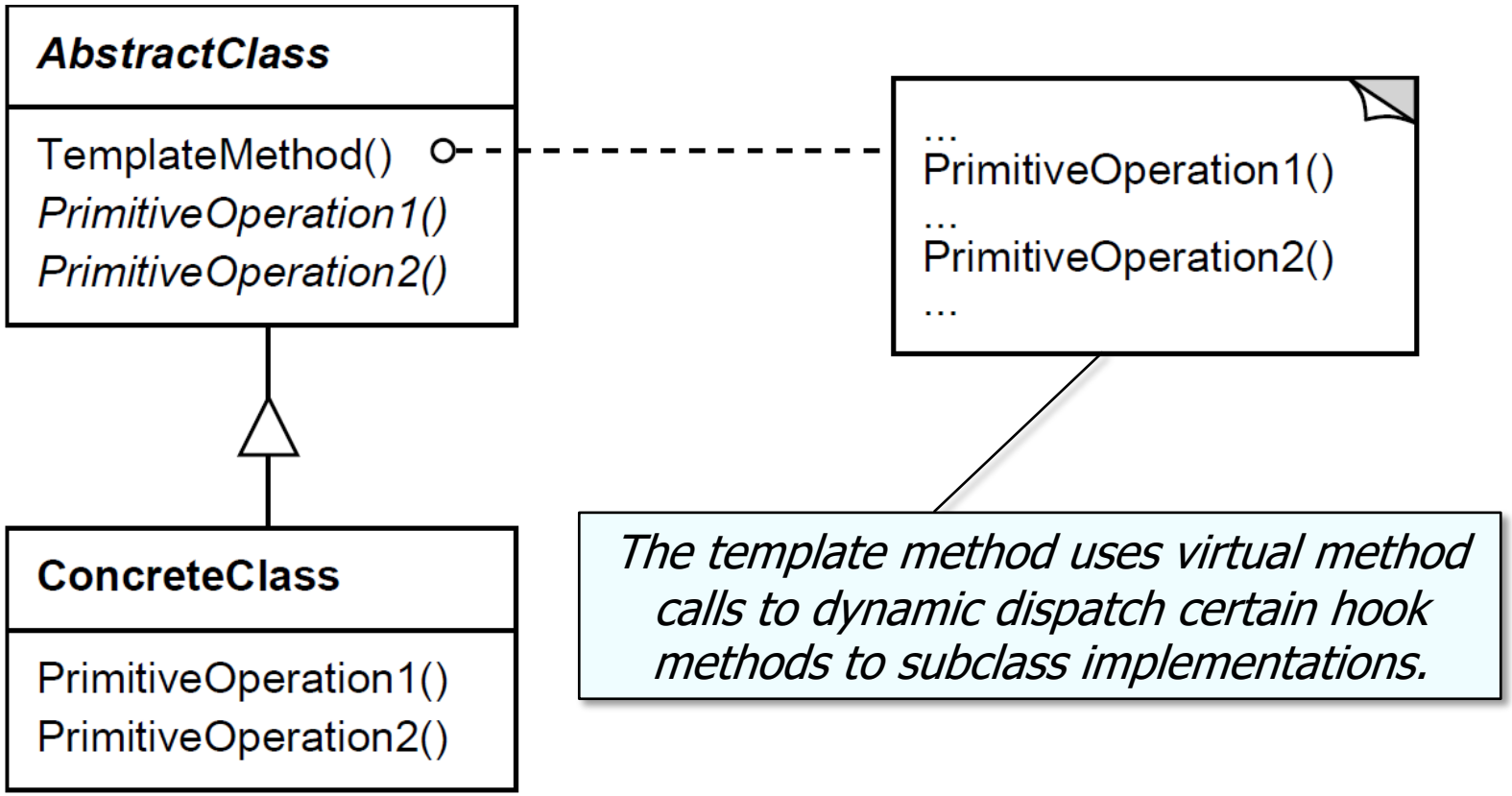


Structure and participants



See wiki.c2.com/?HookMethod

Dynamics



This pattern is "OOP 101" www.stroustrup.com/whatis.pdf

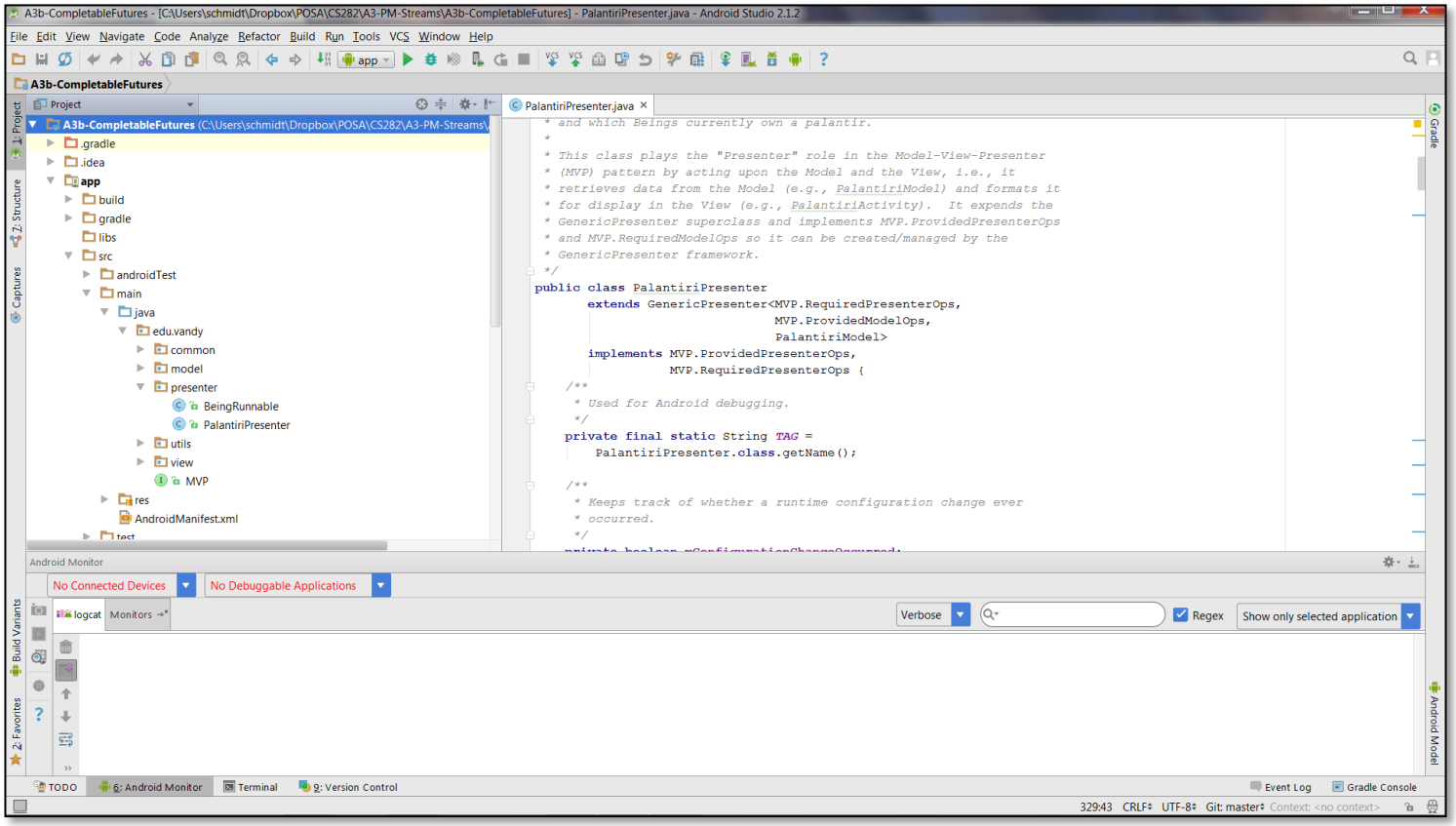
The Template Method Pattern

Implementation in Java

Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Template Method* pattern can be applied to flexibly support multiple operating modes in the expression tree processing app.
- Understand the structure and functionality of the *Template Method* pattern.
- Know how to implement the *Template Method* pattern in Java.



Template Method example in Java

- Allow subclasses to customize certain steps in the input handling algorithm.

```
public abstract class InputHandler {
   ...
   void handleInput() {
     promptUser();
     String input = retrieveInput();
     UserCommand command = makeUserCommand(input);
     executeCommand(command);
   }
```

Template method

See ExpressionTree/CommandLine/src/expressiontree/input

Template Method example in Java

- Allow subclasses to customize certain steps in the input handling algorithm.

```
public abstract class InputHandler {
   ...
   void handleInput() {
     promptUser();
     String input = retrieveInput();
     UserCommand command = makeUserCommand(input);
     executeCommand(command);
   }
```

Hook methods

Template Method example in Java

- Allow subclasses to customize certain steps in the input handling algorithm.

```
public abstract class InputHandler {
   ...
   void handleInput() {
     promptUser();
     String input = retrieveInput();
     UserCommand command = makeUserCommand(input);
     executeCommand(command);
   }

   InputHandler makeHandler(boolean verbose) {
     return verbose ? new VerboseModeInputHandler
                    : new SuccintModeInputHandler;
   }
```

Factory method creates designated concrete classes

This is not the only/best way to define a factory since it's too tightly coupled.

Template Method example in Java

- Allow subclasses to customize certain steps in the input handling algorithm.

```
public class VerboseModeInputHandler extends InputHandler {
   ...
   public UserCommand makeUserCommand(String userInput) {
     return mUserCommandFactory.makeUserCommand
                 (userInput);
   }
```

Specialized hook method

See ExpressionTree/CommandLine/src/expressiontree/input

Template Method example in Java

- Allow subclasses to customize certain steps in the input handling algorithm.

```
public class VerboseModeInputHandler extends InputHandler {
  ...
  public UserCommand makeUserCommand(String userInput) {
    return mUserCommandFactory.makeUserCommand
                (userInput);
  }
```

```
public class SuccinctModeInputHandler extends InputHandler {
  ...
  public UserCommand makeUserCommand(String userInput) {
    return mUserCommandFactory.makeUserCommand
                ("macro " + userInput);
  }
```

Specialized hook method

Template Method example in Java

- Allow subclasses to customize certain steps in the input handling algorithm.

```
public class VerboseModeInputHandler extends InputHandler {
   ...
   public UserCommand makeUserCommand(String userInput) {
     return mUserCommandFactory.makeUserCommand
                 (userInput);
   }


public class SuccinctModeInputHandler extends InputHandler {
   ...
   public UserCommand makeUserCommand(String userInput) {
     return mUserCommandFactory.makeUserCommand
                 ("macro " + userInput);
   }
```

See earlier lesson on "*The Factory Method Pattern*" for **`UserCommandFactory`**.

The Template Method Pattern

Other Considerations

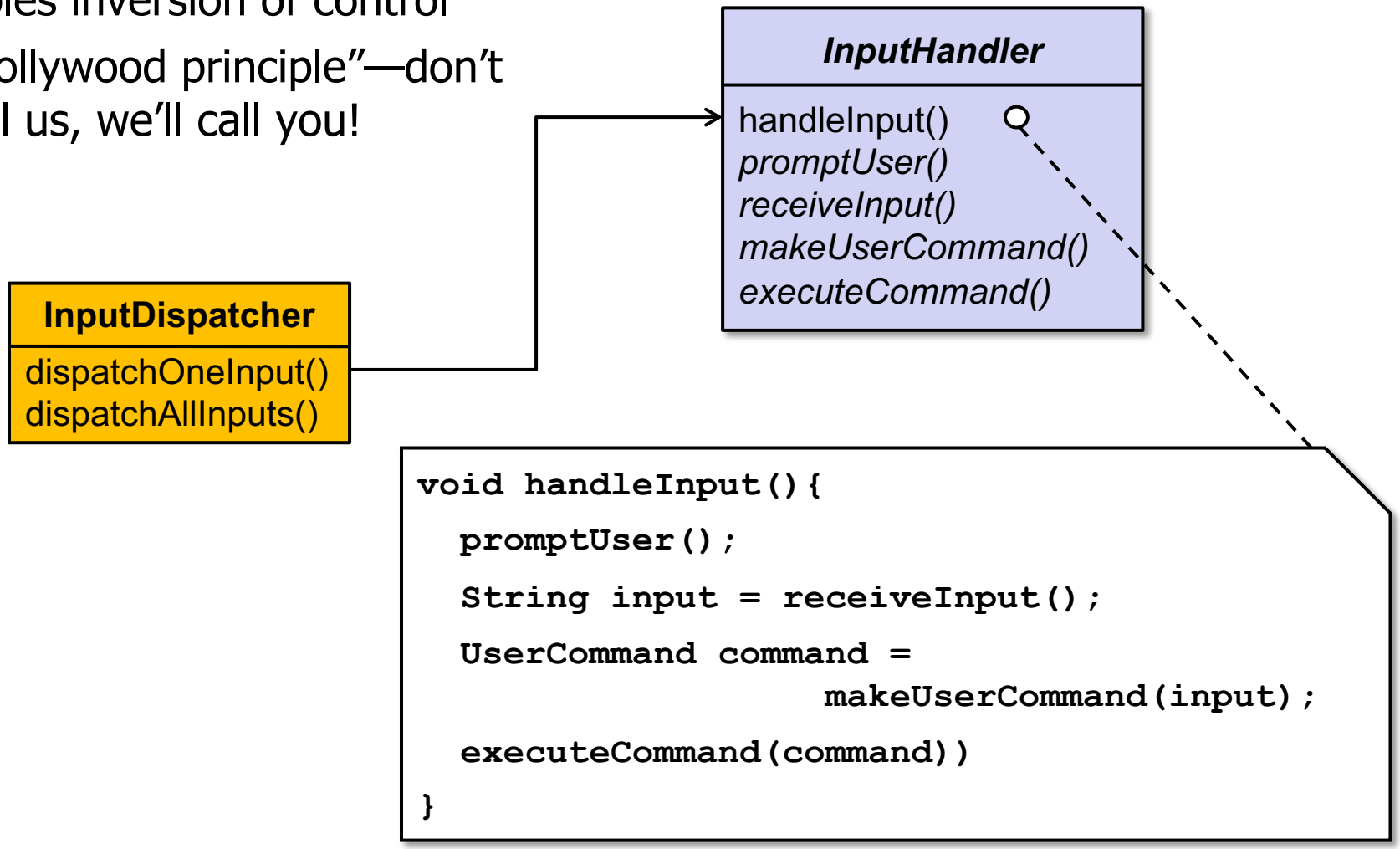
Douglas C. Schmidt

Learning Objectives in This Lesson

- Recognize how the *Template Method* pattern can be applied to flexibly support multiple operating modes in the expression tree processing app.
- Understand the structure and functionality of the *Template Method* pattern.
- Know how to implement the *Template Method* pattern in Java.
- Be aware of other considerations when applying the *Template Method* pattern.

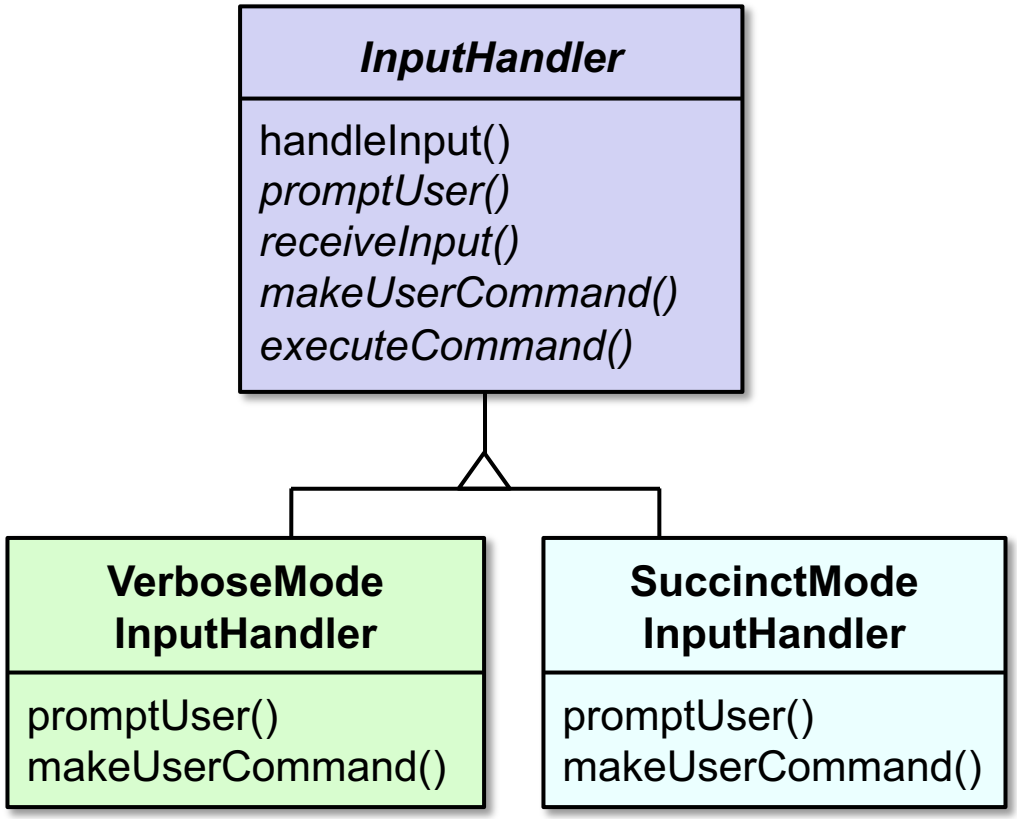
Consequences

+ Enables inversion of control
  - "Hollywood principle"—don't call us, we'll call you!



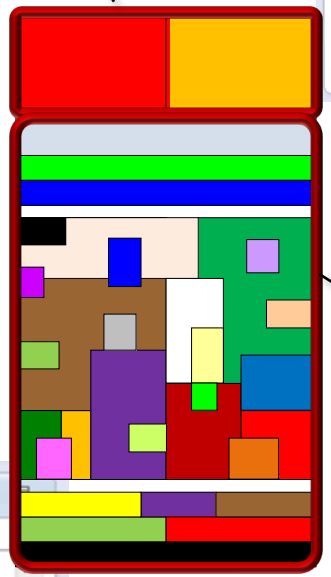
Consequences

+ Overriding rules are enforced via subclassing



Consequences

+ Promotes code reuse by collapsing stovepipes



Variant (nonreusable) code

Common (reusable) code

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)
```

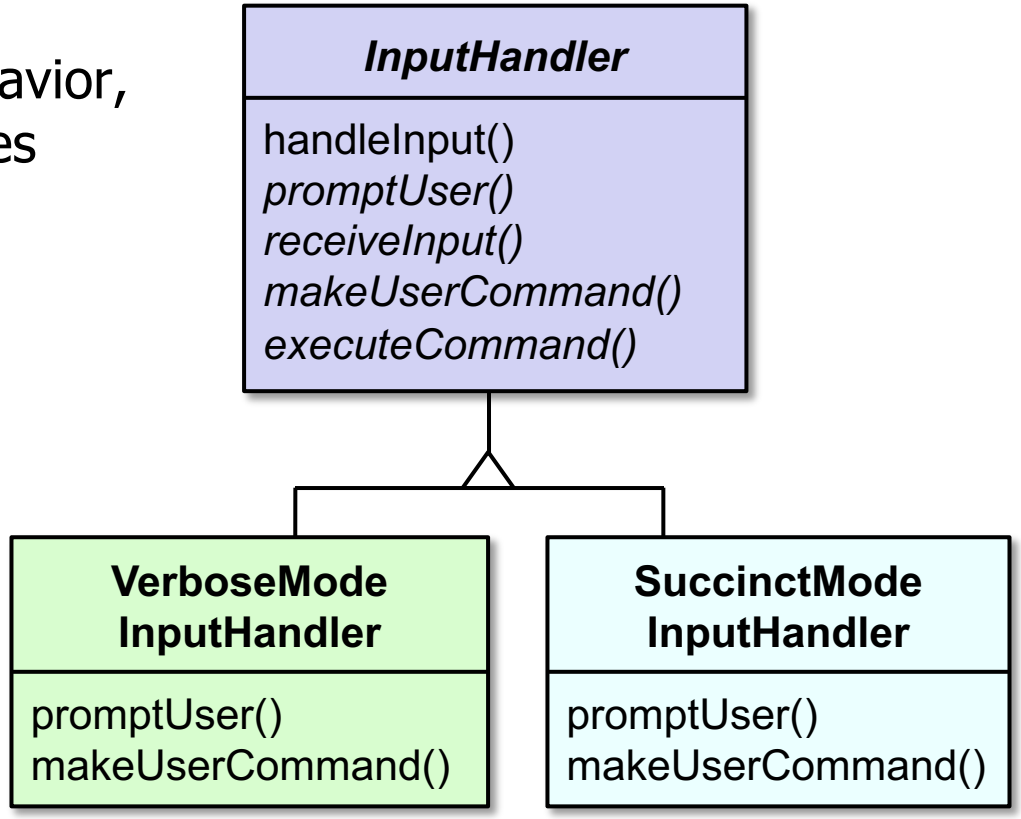
Console

Main [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe

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

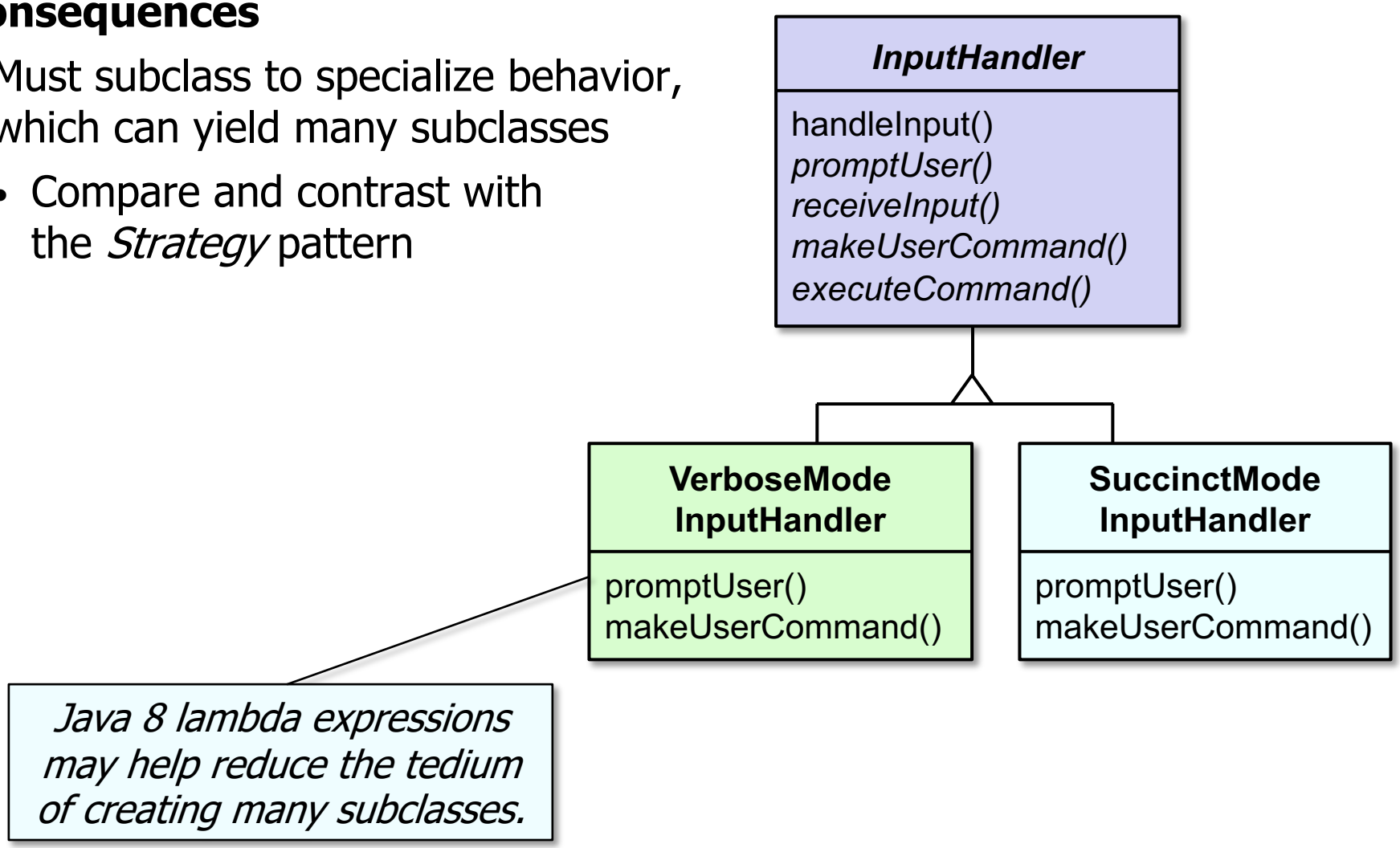
Consequences

– Must subclass to specialize behavior, which can yield many subclasses
  - Compare and contrast with the *Strategy* pattern



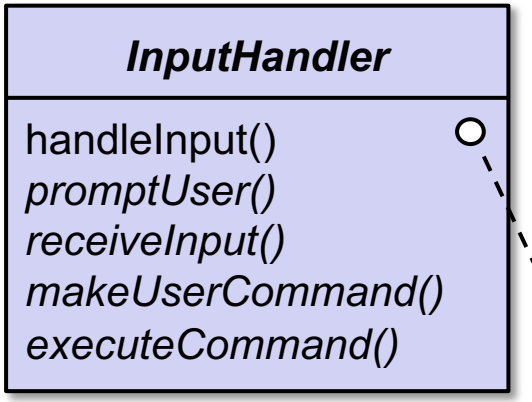
Consequences

– Must subclass to specialize behavior, which can yield many subclasses
  - Compare and contrast with the *Strategy* pattern



Implementation considerations

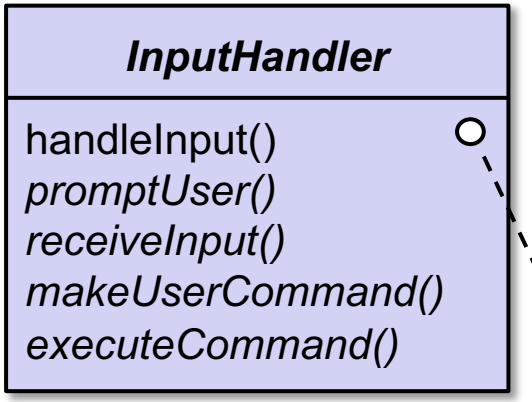
- Virtual vs. non-virtual (final) template method
  - Depends on whether the algorithm embodied by the template method itself may need to change




```
void handleInput(){
  promptUser();
  String input = receiveInput();
  UserCommand command =
                    makeUserCommand(input);
  executeCommand(command))
}
```

Implementation considerations

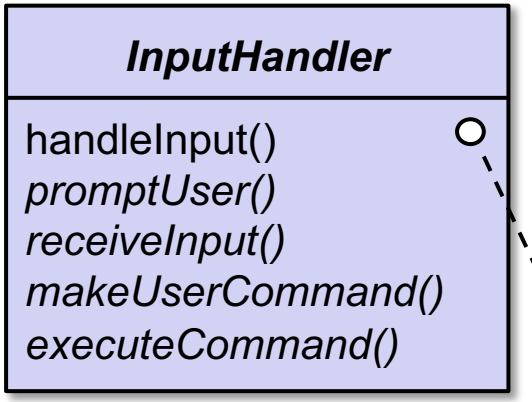
- Few vs. many primitive operations (hook methods)
  - e.g., how much variability's needed in the template method's algorithm?




```
void handleInput(){
  promptUser();
  String input = receiveInput();
  UserCommand command =
                    makeUserCommand(input);
  executeCommand(command))
}
```

Implementation considerations

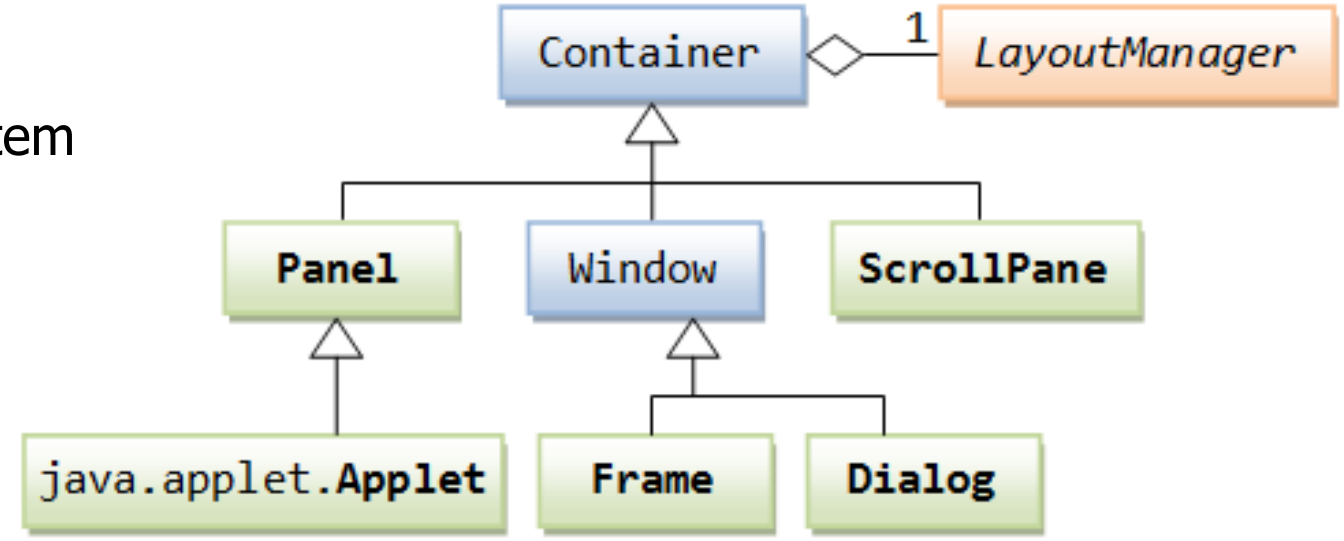
- Naming conventions
  - e.g., do*() vs. make*() vs. on*() prefixes




```
void handleInput(){
  promptUser();
  String input = receiveInput();
  UserCommand command =
                    makeUserCommand(input);
  executeCommand(command))
}
```

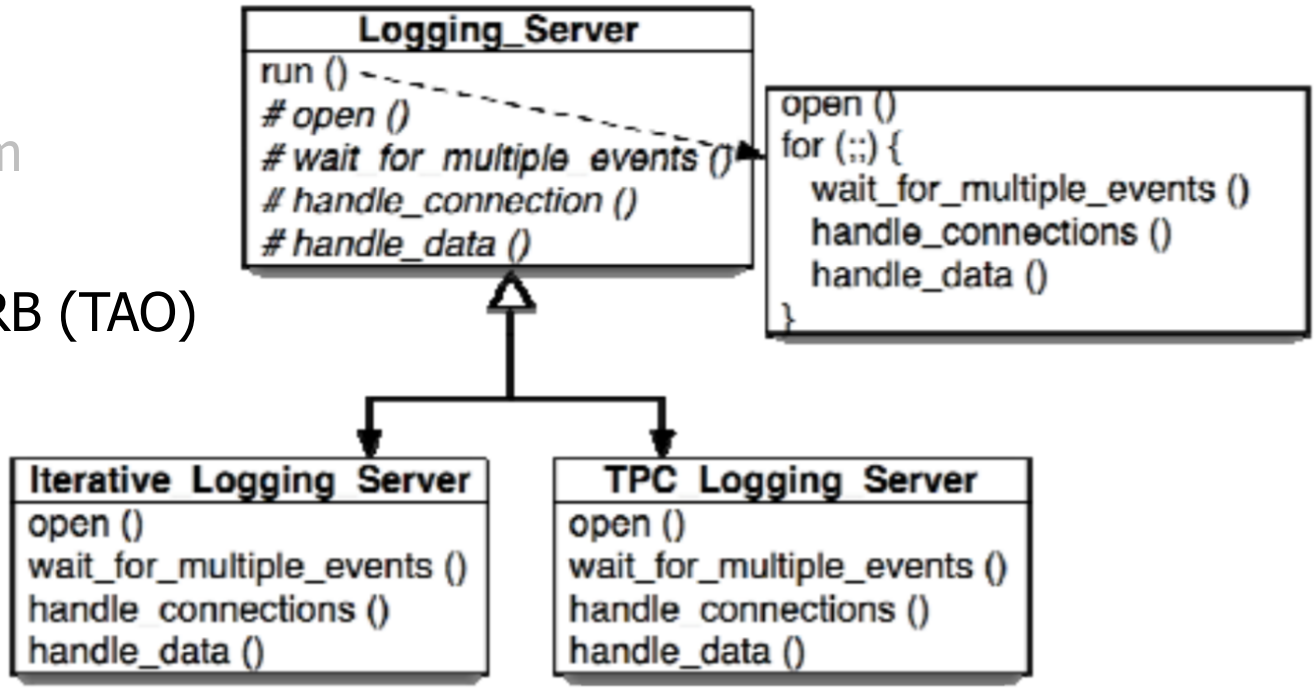
Known uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit



Known uses

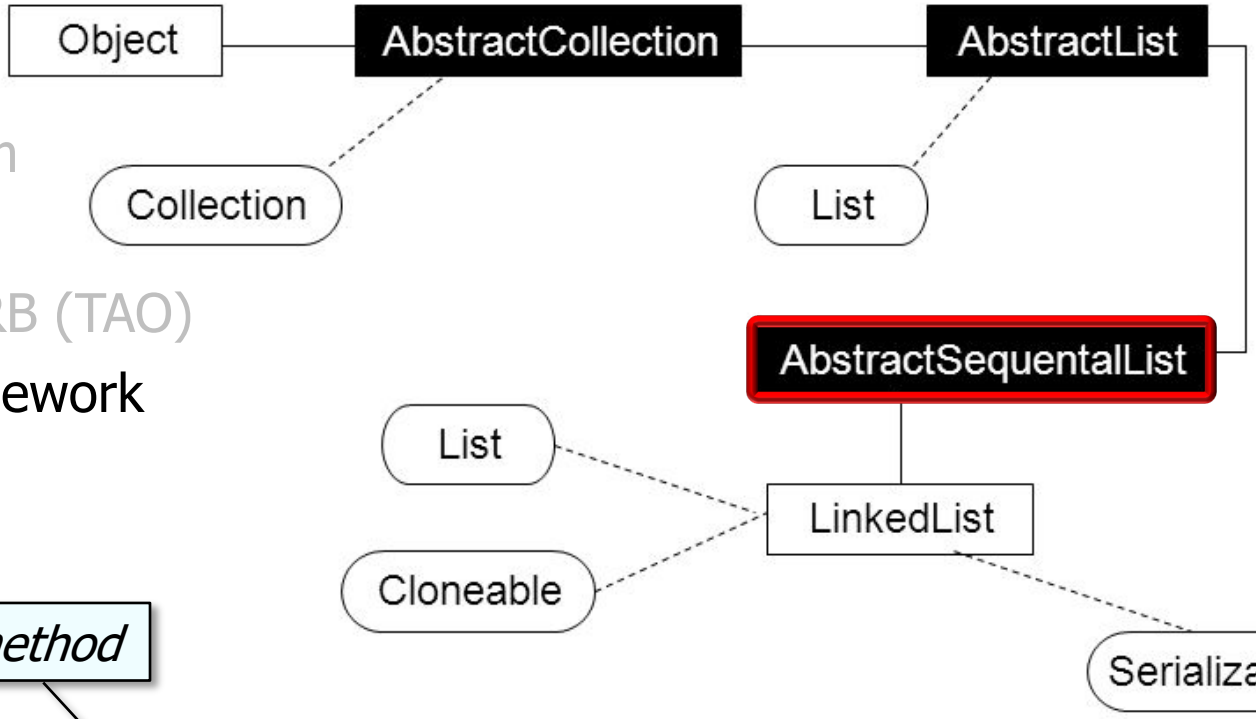
- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- ACE and The ACE ORB (TAO)



Template Method GoF Class Behavioral

Known uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- ACE and The ACE ORB (TAO)
- Java Collections framework



Template method

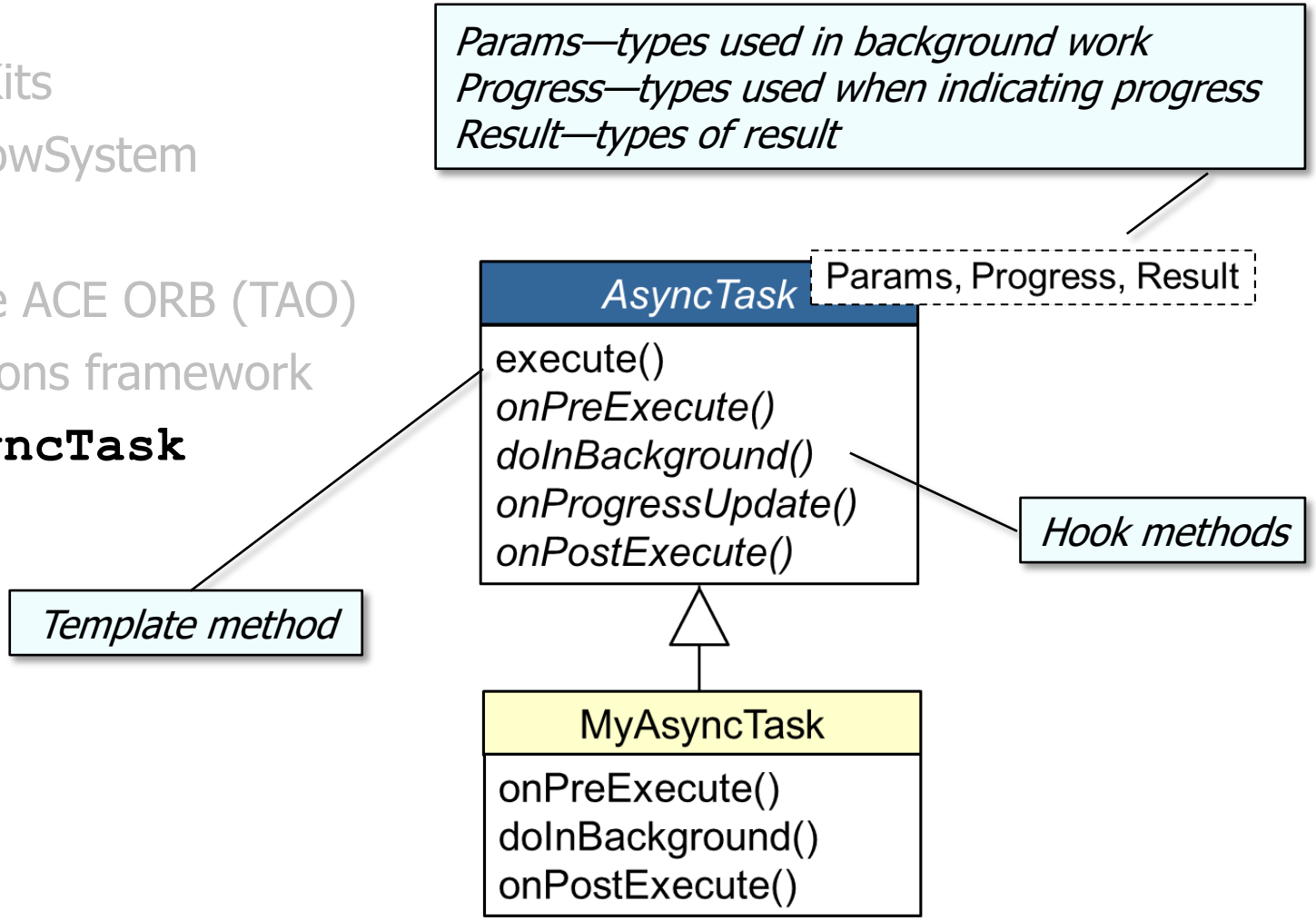
```
public E set(int index, E element) {
    try {
      ListIterator<E> e = listIterator(index);
      ...
```

Hook method

See refactoring.guru/design-patterns/template-method/java/example

Known uses

- InterViews Kits
- ET++ WindowSystem
- AWT Toolkit
- ACE and The ACE ORB (TAO)
- Java Collections framework
- Android **`AsyncTask`** framework



See developer.android.com/reference/android/os/AsyncTask.html

Comparing Strategy With Template Method

Strategy

+ Provides for clean separation between components via "black-box" interfaces
+ Allows for strategy composition at runtime
+ Supports flexible mixing and matching of features
– May yield many strategy classes
– Incurs forwarding overhead

Comparing Strategy With Template Method

Strategy

+ Provides for clean separation between components via "black-box" interfaces
+ Allows for strategy composition at runtime
+ Supports flexible mixing and matching of features
– May yield many strategy classes
– Incurs forwarding overhead

Template Method

+ No explicit forwarding necessary
+ May be easier for small use cases
– Close coupling between subclass(es) and super class
– Inheritance hierarchies are static and cannot be reconfigured at runtime
– Adding features via inheritance may yield combinatorial subclass explosion
  – Beware overusing inheritance since it's not always the best choice.
  – Deep inheritance hierarchies in an app are a red flag.

We selected *Template Method* for our case study since it's a simple use case.

Comparing Strategy With Template Method

Strategy

+ Provides for clean separation between components via "black-box" interfaces
+ Allows for strategy composition at runtime
+ Supports flexible mixing and matching of features
– May yield many strategy classes
– Incurs forwarding overhead

Template Method

+ No explicit forwarding necessary
+ May be easier for small use cases
– Close coupling between subclass(es) and super class
– Inheritance hierarchies are static and cannot be reconfigured at runtime
– Adding features via inheritance may yield combinatorial subclass explosion
  – Beware overusing inheritance since it's not always the best choice.
  – Deep inheritance hierarchies in an app are a red flag.

Template Method and Strategy are often treated as "pattern complements" that provide alternative solutions to related design problems.

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

Summary of the Template Method Pattern

- *Template Method* enables controlled variability of steps in the **`InputHandler`** algorithm for processing multiple operating modes, which enhances reuse.