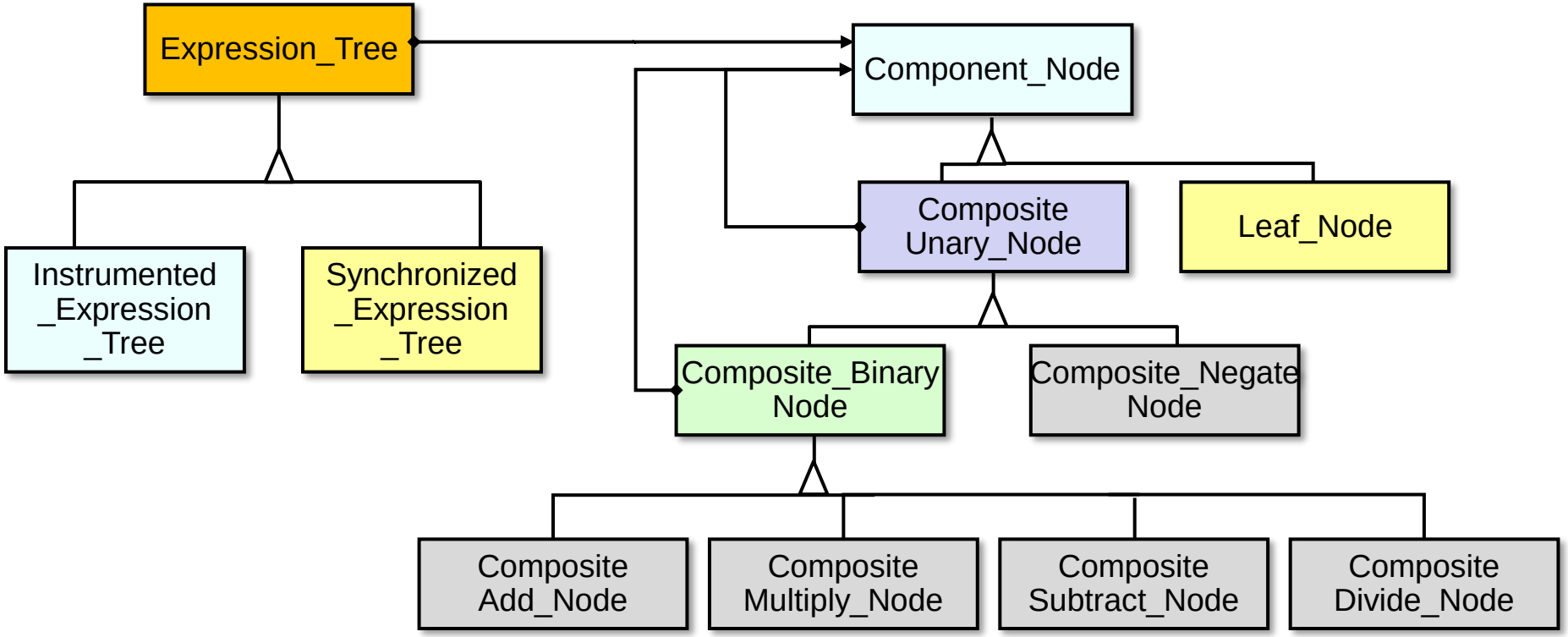
The Bridge Pattern

Motivating Example

Douglas C. Schmidt

Learning Objectives in This Lesson

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

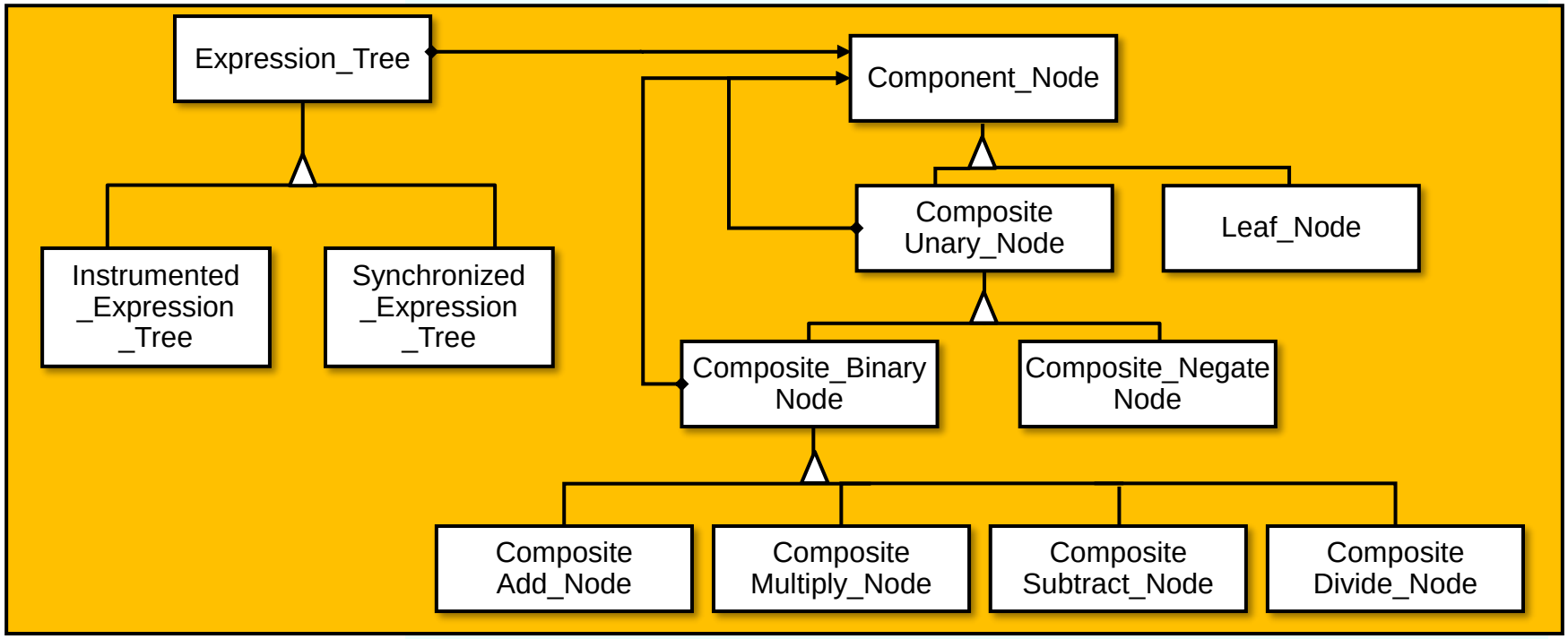


Douglas C. Schmidt

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

A Pattern for Binding One of Many Variations

***Purpose**: Decouple the expression tree programming API from its behavior & implementation to enable transparent extensibility.*



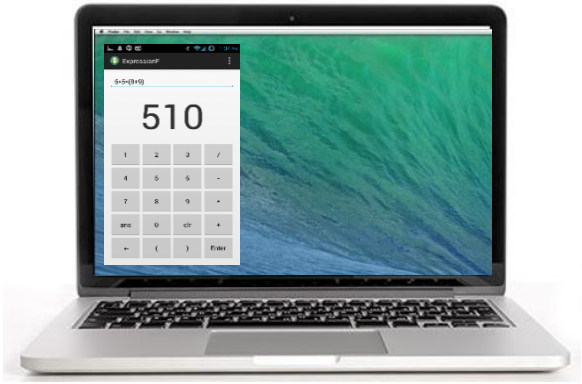
Bridge minimizes coupling between clients, abstractions, & implementations.

Context: OO Expression Tree Processing App

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

Context: OO Expression Tree Processing App

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



Context: OO Expression Tree Processing App

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

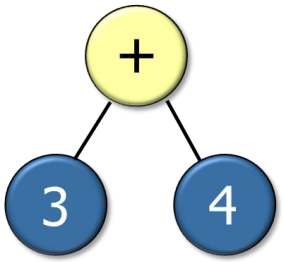
Problem: Minimizing Impact of Variability

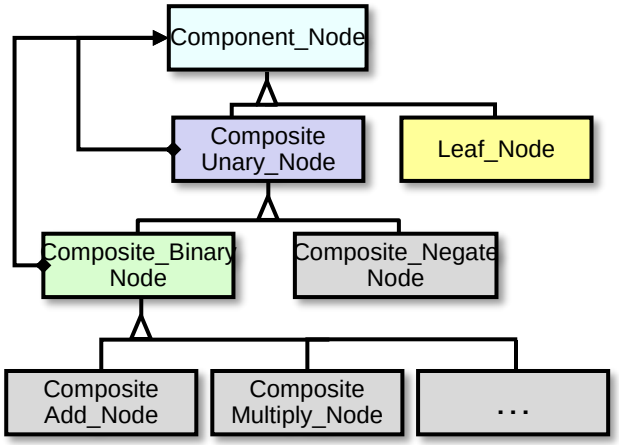
- Tightly coupling app components to a particular environment has drawbacks.

Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks.
  - Suboptimal implementations for a given context

```
Component_Node node =
   new Composite_Add_Node
      (new Leaf_Node(3),
       new Leaf_Node(4));
```

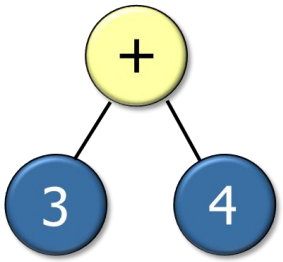


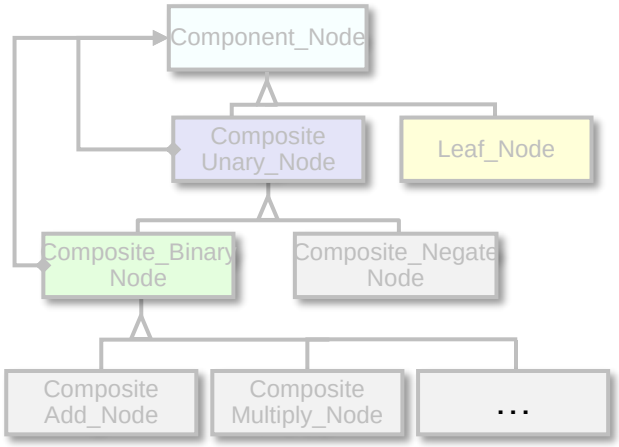


Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks.
  - Suboptimal implementations for a given context

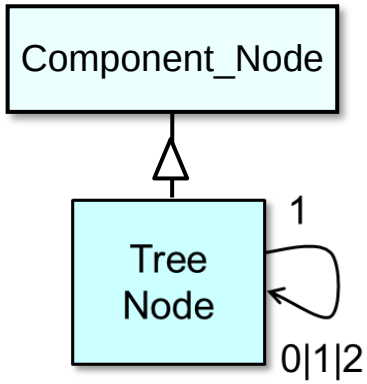
```
Component_Node node =
  new Composite_Add_Node
    (new Leaf_Node(3),
     new Leaf_Node(4));
```





VS.

```
Component_Node node =
  new Tree_Node
    ('+',
     new Tree_Node(3),
     new Tree_Node(4));
```

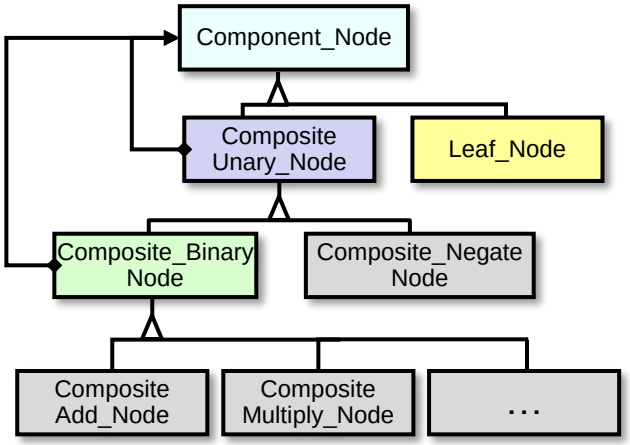


Problem: Minimizing Impact of Variability

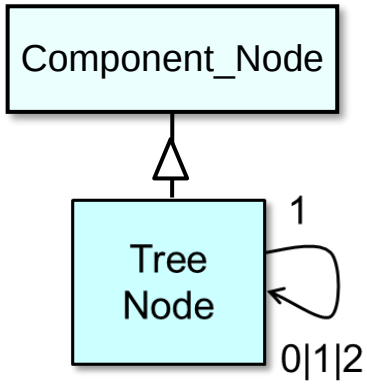
- Tightly coupling app components to a particular environment has drawbacks.
  - Suboptimal implementations for a given context

```
Component_Node node =
   new Composite_Add_Node
      (new Leaf_Node(3),
       new Leaf_Node(4));
```

Different implementations have different time/space trade-offs



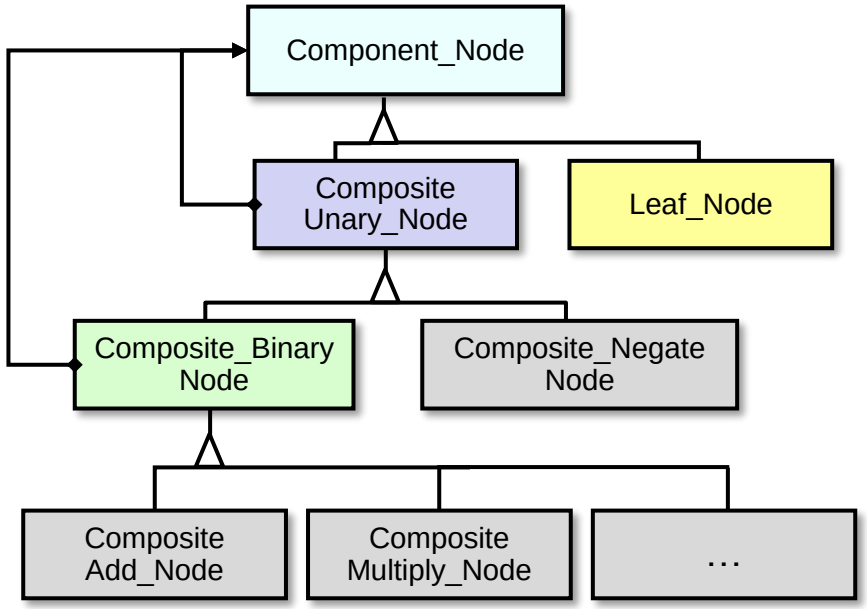

```
Component_Node node =
   new Tree_Node
      ('+',
       new Tree_Node(3),
       new Tree_Node(4));
```



We should be able to change implementations without breaking client code.

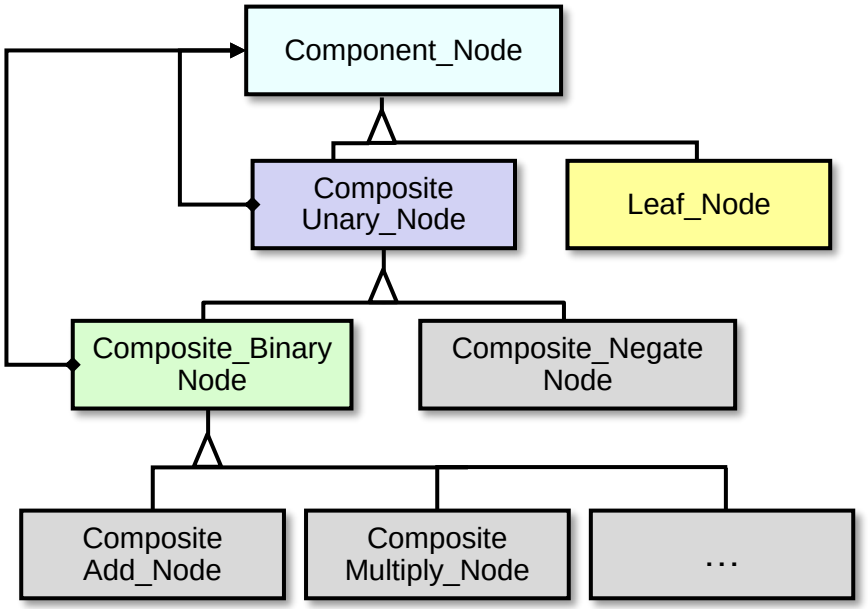
Problem: Minimizing Impact of Variability

- Tightly coupling app components to a particular environment has drawbacks.
  - Suboptimal implementations for a given context
  - Hard to change services transparently



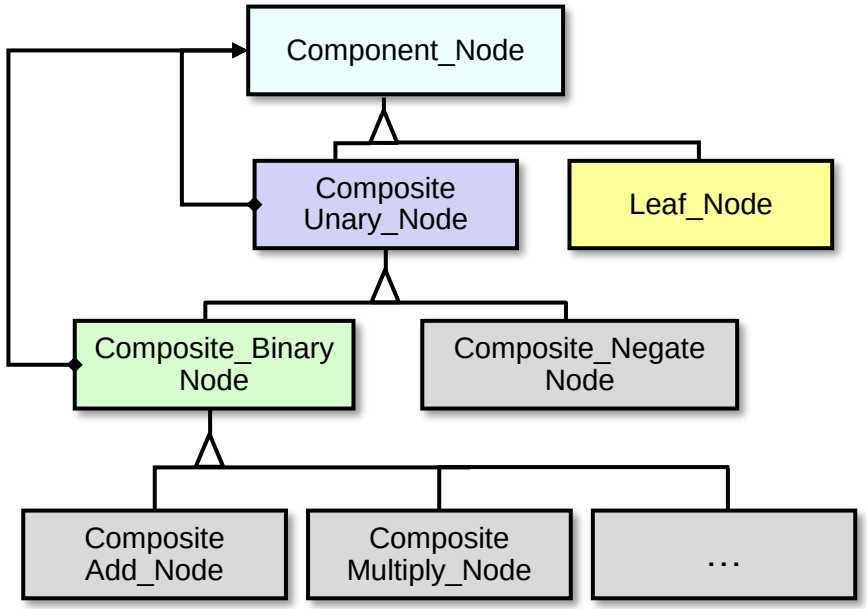
Problem: Minimizing Impact of Variability

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



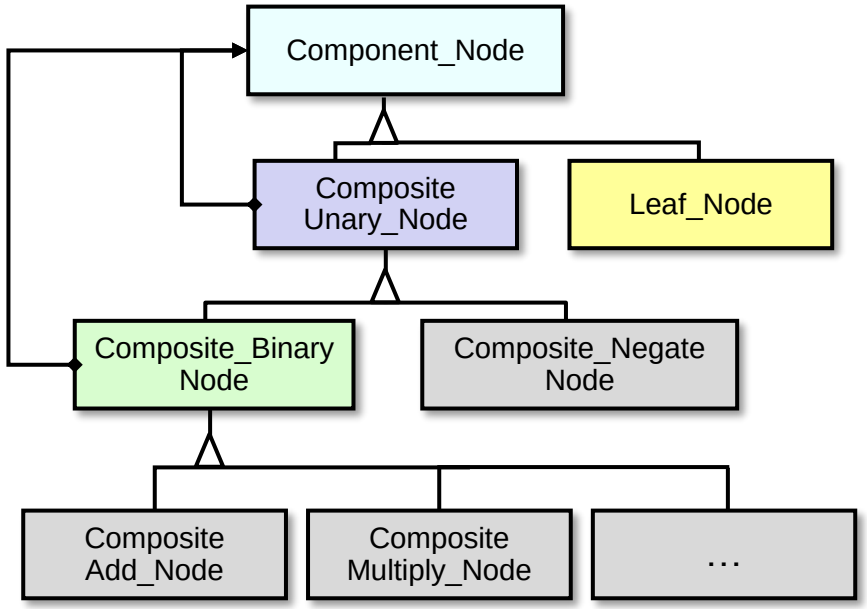
Problem: Minimizing Impact of Variability

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



Problem: Minimizing Impact of Variability

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



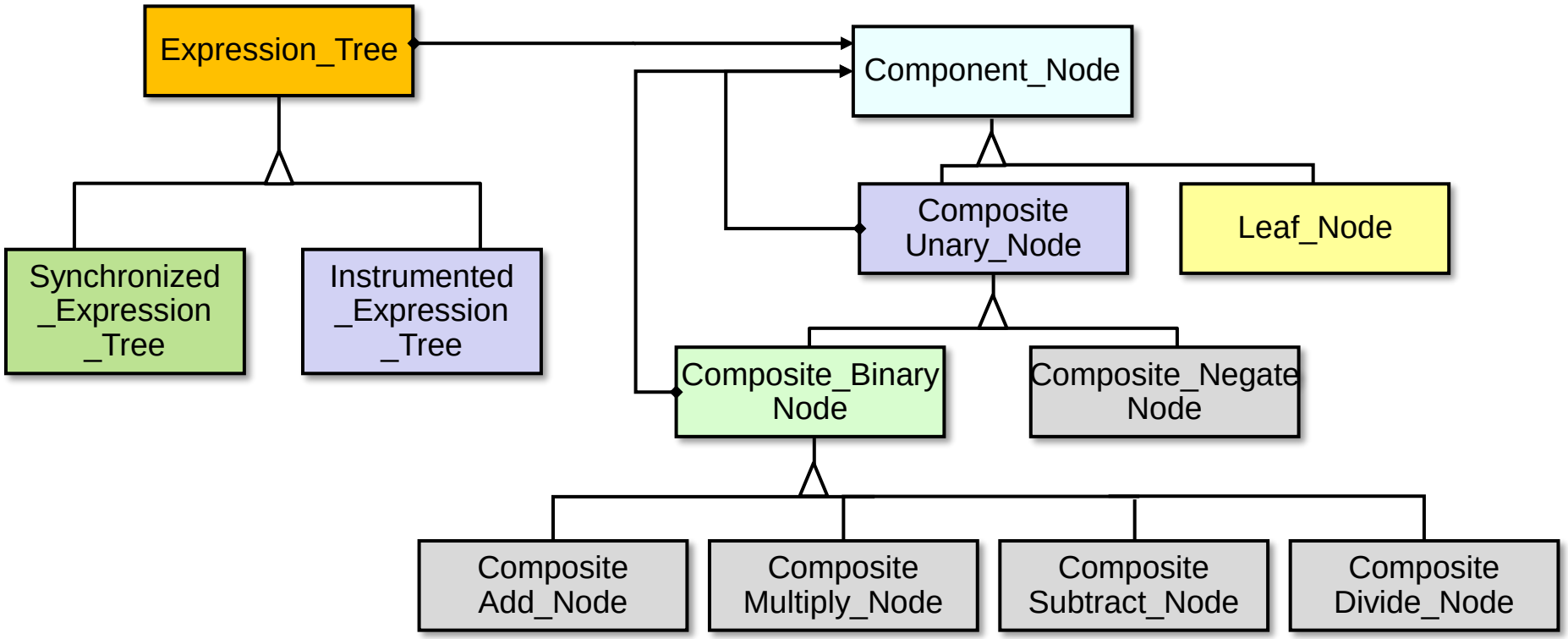
We should be able to enhance the service without breaking its implementation.

Solution: Separate Abstraction & Implementation

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

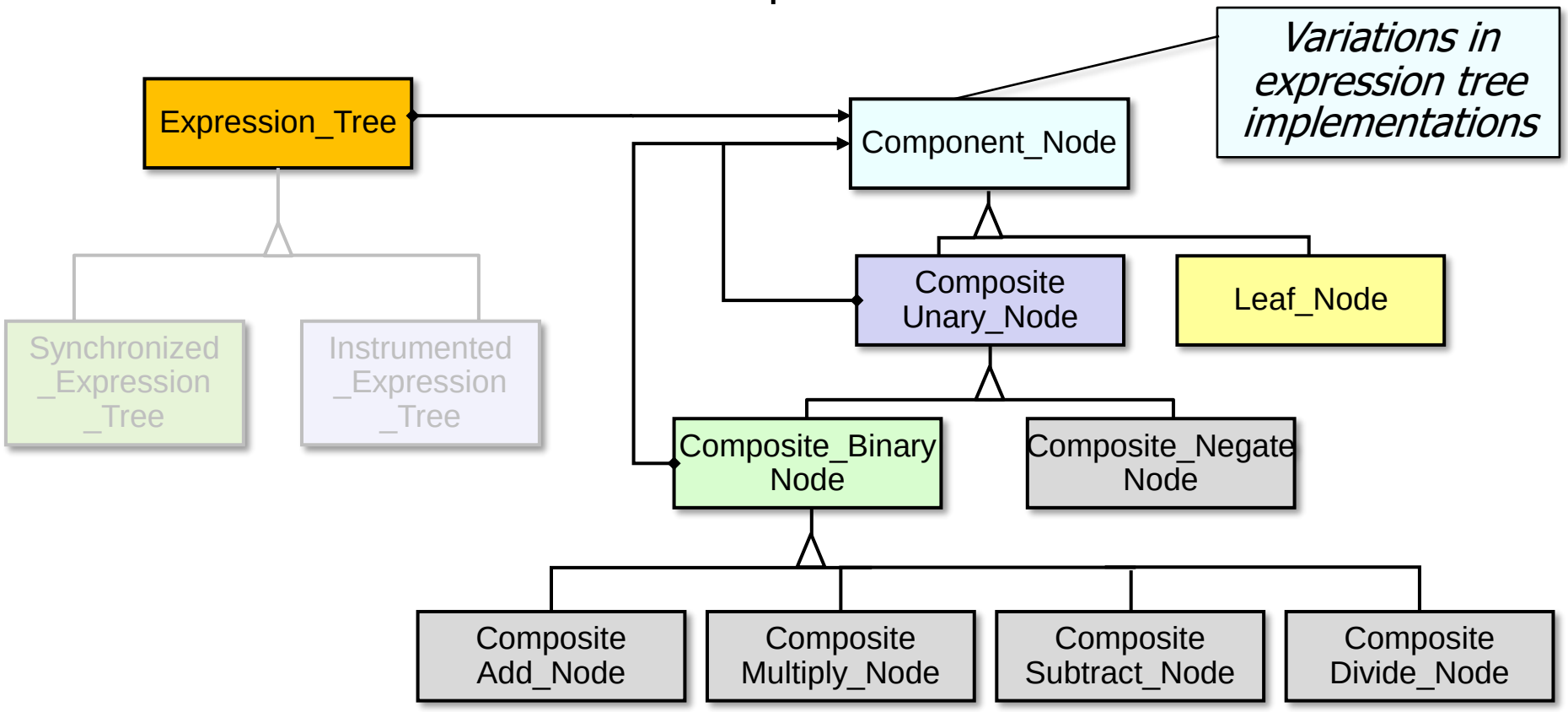
Solution: Separate Abstraction & Implementation

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



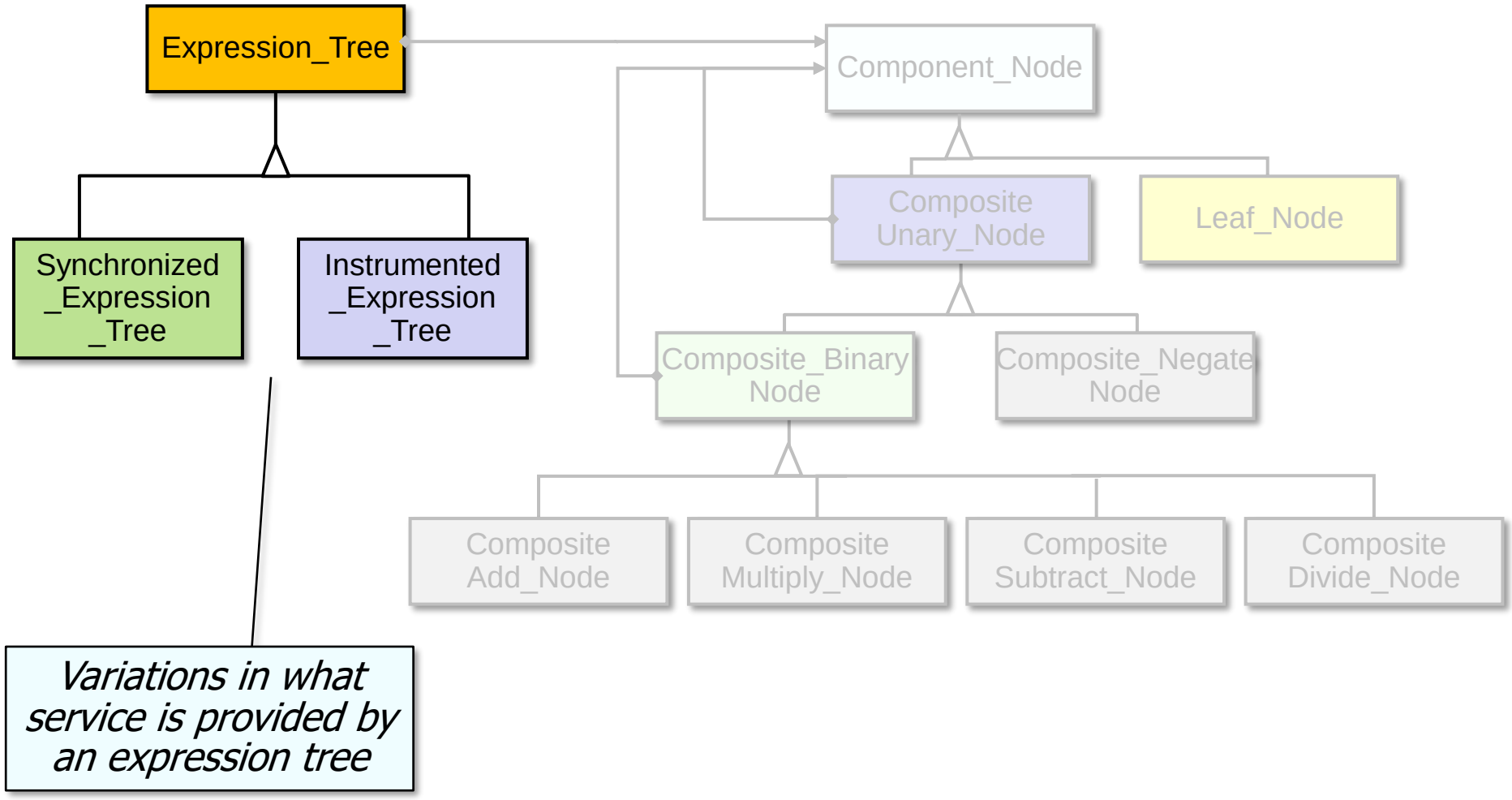
Solution: Separate Abstraction & Implementation

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



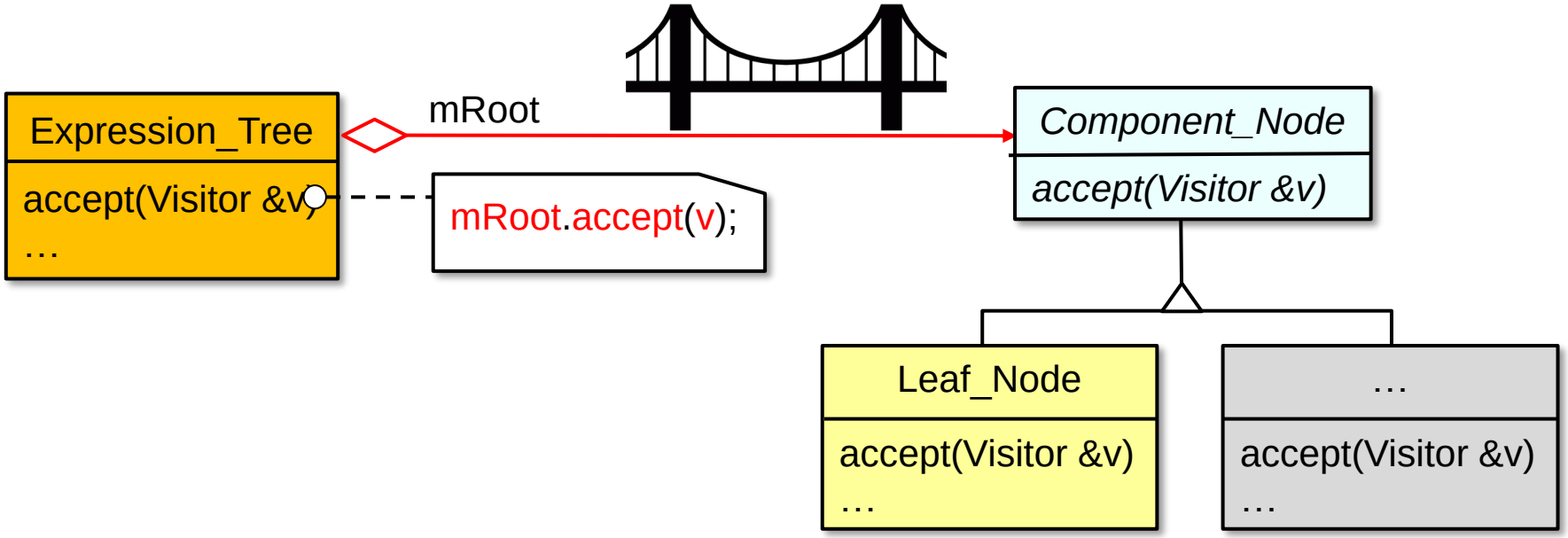
Solution: Separate Abstraction & Implementation

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



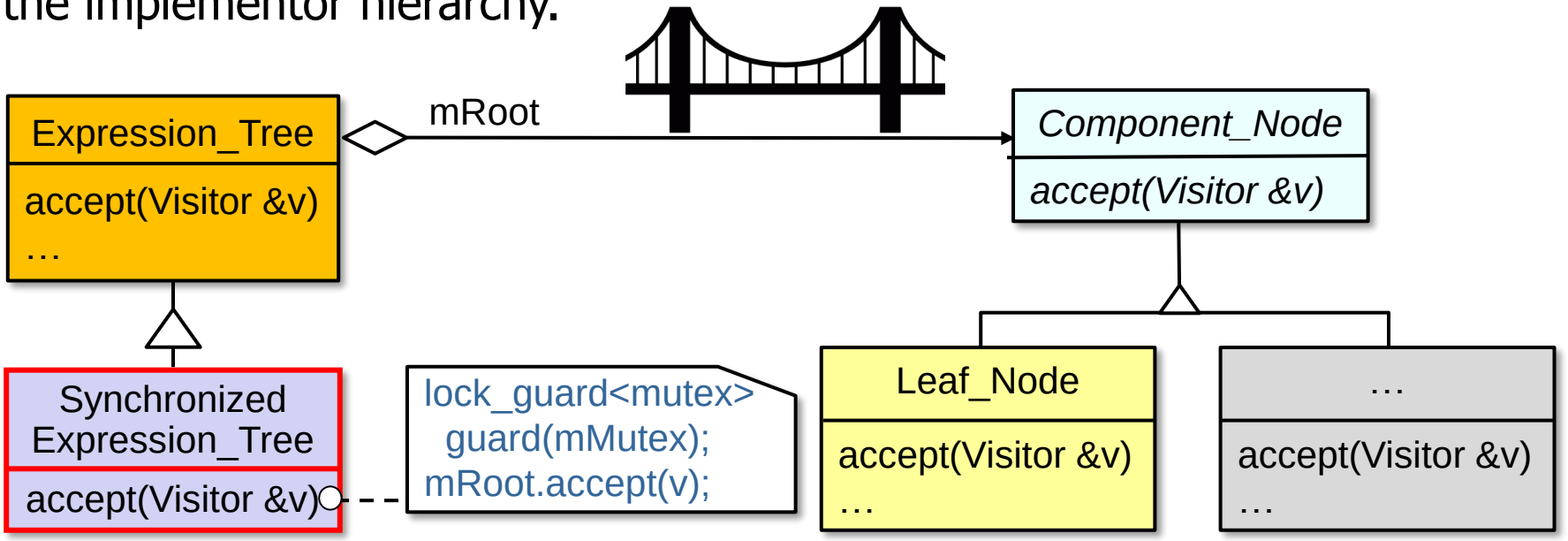
Solution: Separate Abstraction & Implementation

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



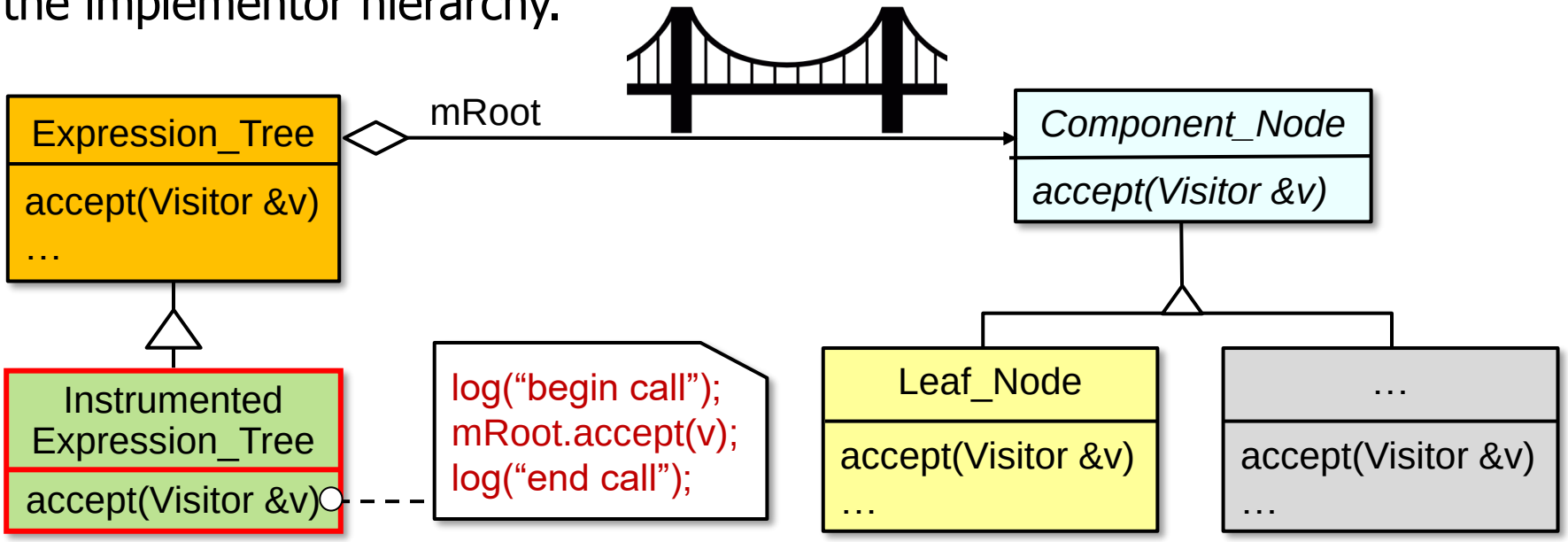
Solution: Separate Abstraction & Implementation

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



Solution: Separate Abstraction & Implementation

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



Expression_Tree Class Overview

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

Class methods

```
                Expression_Tree(Component_Node *root)
           bool is_null()
            int item()
Expression_Tree left()
Expression_Tree right()
           void accept(Visitor &visitor)
       iterator begin(const string &traversal_order)
       iterator end(const string &traversal_order)
```

See upcoming lessons on *Factory Method*, *Iterator*, & *Visitor* patterns.

Expression_Tree Class Overview

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

Class methods

Pass in the root of the implementor hierarchy

```
                Expression_Tree(Component_Node *root)
           bool is_null()
            int item()
Expression_Tree left()
Expression_Tree right()
           void accept(Visitor &visitor)
       iterator begin(const string &traversal_order)
       iterator end(const string &traversal_order)
```

Expression_Tree Class Overview

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

Class methods

Forward to implementor hierarchy

```
Expression_Tree(Component_Node *root)
bool is_null()
int item()
Expression_Tree left()
Expression_Tree right()
void accept(Visitor &visitor)
iterator begin(const string &traversal_order)
iterator end(const string &traversal_order)
```

Expression_Tree Class Overview

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

Class methods

```
Expression_Tree(Component_Node *root)
bool is_null()
int item()
Expression_Tree left()
Expression_Tree right()
void accept(Visitor &visitor)
iterator begin(const string &traversal_order)
iterator end(const string &traversal_order)
```

Plays essential role in the *Iterator* & *Visitor* patterns.

See upcoming lessons on "*The Iterator Pattern*" & "*The Visitor Pattern*."

Expression_Tree Class Overview

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

Class methods

```
                Expression_Tree(Component_Node *root)
           bool is_null()
            int item()
Expression_Tree left()
Expression_Tree right()
           void accept(Visitor &visitor)
       iterator begin(const string &traversal_order)
       iterator end(const string &traversal_order)
```

Factory methods creates iterators

See upcoming lessons on *Factory Method*, *Iterator*, & *Strategy* patterns.

Expression_Tree Class Overview

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

Class methods

```
                Expression_Tree(Component_Node *root)
           bool is_null()
            int item()
Expression_Tree left()
Expression_Tree right()
           void accept(Visitor &visitor)
       iterator begin(const string &traversal_order)
       iterator end(const string &traversal_order)
```

- **Commonality**: provides a common interface for expression tree operations
- **Variability**: component nodes will vary depending on user input expressions; iterator behavior can vary; & expression tree itself can vary