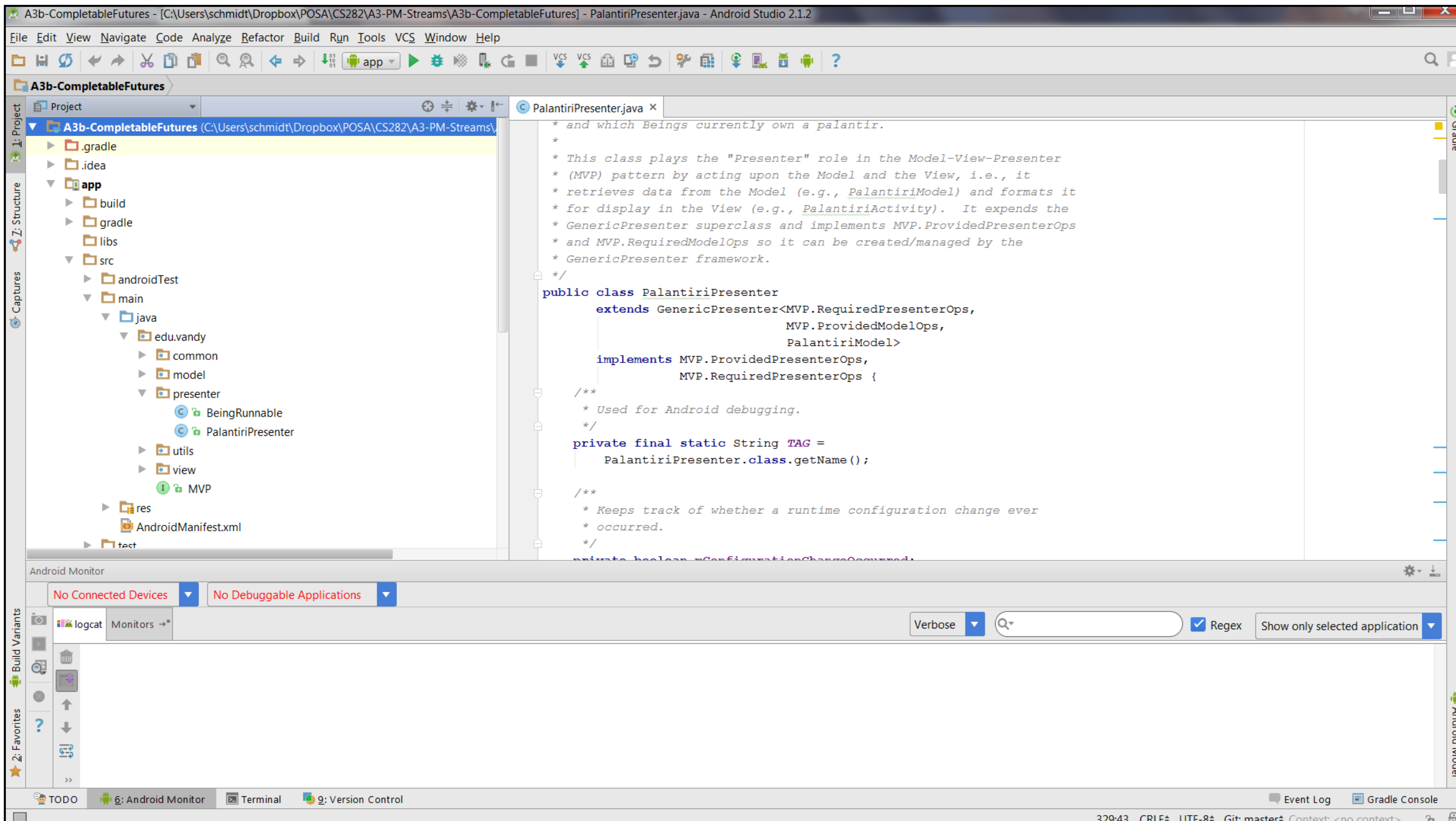
The Bridge Pattern

Implementation in C++

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.
- Understand the structure & functionality of the *Bridge* pattern.
- Know how to implement the *Bridge* pattern in C++.



Douglas C. Schmidt

Implementing the Bridge Pattern in C++

Bridge example in C++

- Separate expression tree abstraction from composite implementor hierarchy.

```
class Expression_Tree {
  Refcounter<Component_Node> root_;

  Expression_Tree(Component_Node *root)
    : root_ (root) {}


  ...

  void accept(Visitor &v) { root_->accept(v); }

}
```

Bridge example in C++

- Separate expression tree abstraction from composite implementor hierarchy.

```
class Expression_Tree {
  Refcounter<Component_Node> root_;

  Expression_Tree(Component_Node *root)
    : root_ (root) {}


  ...

  void accept(Visitor &v) { root_->accept(v); }

}
```

Stores root of composite implementor hierarchy

Bridge example in C++

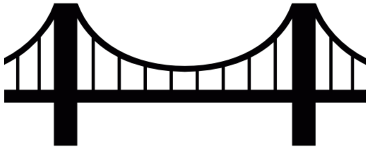
- Separate expression tree abstraction from composite implementor hierarchy.

```
class Expression_Tree {
  Refcounter<Component_Node> root_;

  Expression_Tree(Component_Node *root)
    : root_ (root) {}

  ...

  void accept(Visitor &v) { root_->accept(v); }

}
```

Pass in root of composite implementor hierarchy

Bridge example in C++

- Separate expression tree abstraction from composite implementor hierarchy.



```
class Expression_Tree {
  Refcounter<Component_Node> root_;

  Expression_Tree(Component_Node *root)
    : root_ (root) {}


  ...

  void accept(Visitor &v) { root_->accept(v); }

}
```

Abstraction forwards to implementor via `root_`

See upcoming lessons on *Iterator* & *Visitor* patterns.

Bridge example in C++

- Separate expression tree abstraction from composite implementor hierarchy.

```
class Instrumented_Expression_Tree
      : public Expression_Tree {
  void accept(Visitor &v) {
    log("starting accept() call" ...);
    Expression_Tree.accept(v);
    log("finished accept() call" ...);
  }
  ...
```

Bridge example in C++

- Separate expression tree abstraction from composite implementor hierarchy.

```
class Instrumented_Expression_Tree
      : public Expression_Tree {
  void accept(Visitor &v) {
    log("starting accept() call" ...);
    Expression_Tree.accept(v);
    log("finished accept() call" ...);
  }
  ...
```

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

See www.javacodegeeks.com/2015/10/wrap-around-design-pattern-in-java8.html

Bridge example in C++

- Separate expression tree abstraction from composite implementor hierarchy.

```
class Instrumented_Expression_Tree
      : public Expression_Tree {
  void accept(Visitor &v) {
    log("starting accept() call" ...);
    Expression_Tree.accept(v);
    log("finished accept() call" ...);
  }
  ...

class Synchronized_Expression_Tree
      : public Expression_Tree {
  void accept(Visitor &v) {
    lock_guard<std::mutex> guard (mMutex);
    Expression_Tree.accept(v);
  }
  ...
```

Bridge example in C++

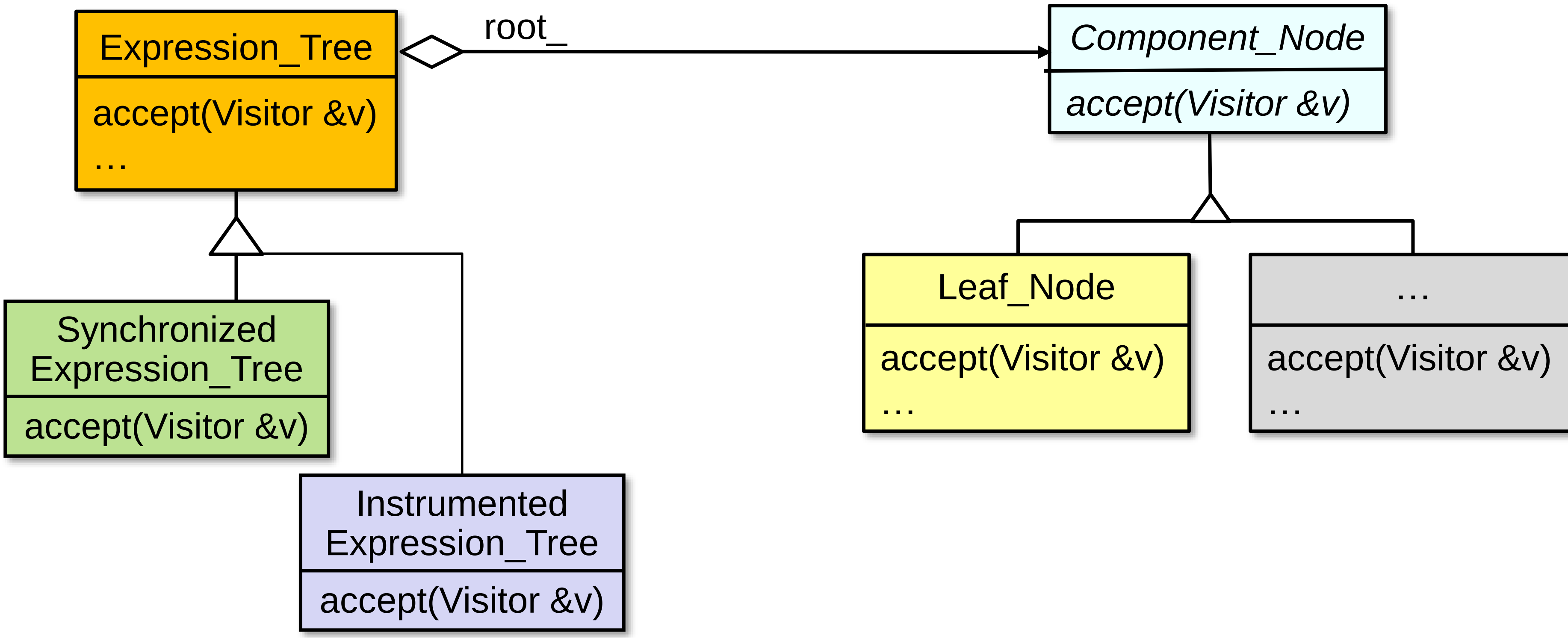
- Separate expression tree abstraction from composite implementor hierarchy.

```
class Instrumented_Expression_Tree
      : public Expression_Tree {
  void accept(Visitor &v) {
    log("starting accept() call" ...);
    Expression_Tree.accept(v);
    log("finished accept() call" ...);
  }
  ...

class Synchronized_Expression_Tree
      : public Expression_Tree {
  void accept(Visitor &v) {
    lock_guard<std::mutex> guard (mMutex);
    Expression_Tree.accept(v);
  }
  ...
```

Synchronize the call to accept()

Bridge example in C++

- Separate expression tree abstraction from composite implementor hierarchy.

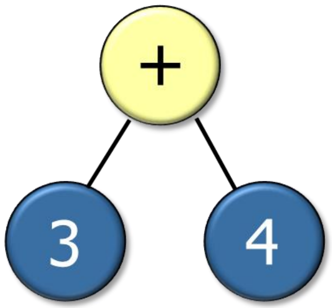


Changes in service behavior don't affect implementor hierarchy & vice versa.

Bridge example in C++

- Encapsulate sources of variability in expression tree construction & use.

```
Expression_Tree expr_tree
  (new Composite_Add_Node
    (new Leaf_Node(3),
     new Leaf_Node(4)));
```

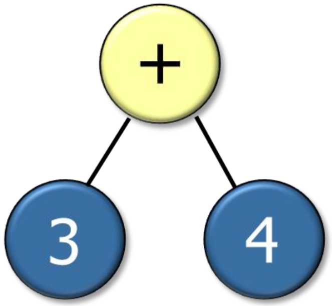


Hide use of complex recursive *Composite* internal structure behind a stable *Bridge* API

Bridge example in C++

- Encapsulate sources of variability in expression tree construction & use.

```
Expression_Tree expr_tree
  (new Composite_Add_Node
    (new Leaf_Node(3),
     new Leaf_Node(4)));
```



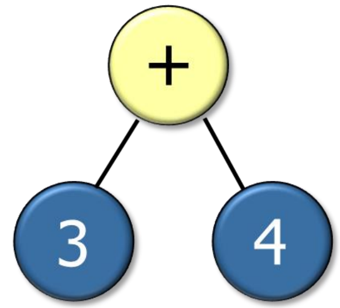
Replace *Composite* implementation with `Tree_Node` implementation

```
Expression_Tree expr_tree
  (new Tree_Node
    ('+',
     new Tree_Node(3),
     new Tree_Node(4)));
```

Bridge example in C++

- Encapsulate sources of variability in expression tree construction & use.

```
Expression_Tree expr_tree
   (make_expression_tree("3+4"));
```



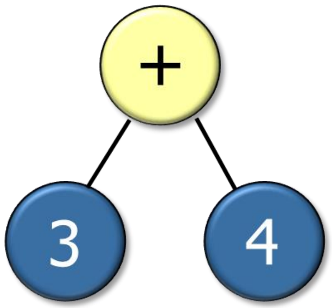
We can apply a creational pattern to reduce client dependencies on implementation variability.

See upcoming lessons on the *Interpreter* & *Builder* patterns.

Bridge example in C++

- Encapsulate sources of variability in expression tree construction & use.

```
for (auto it = expr_tree.begin(order);
     it != expr_tree.end(order);
     ++it)
  do_something_with_each_node(*it);
```



Regardless of which implementation was used, we can iterate thru all tree elements without concern for how the tree is structured internally.

Factory Method, *Composite*, *Iterator*, *Strategy*, & *Visitor* are also relevant here.

The Bridge Pattern

Other Considerations

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.
- Understand the structure & functionality of the *Bridge* pattern.
- Know how to implement the *Bridge* pattern in C++.
- Be aware of other considerations when applying the *Bridge* pattern.