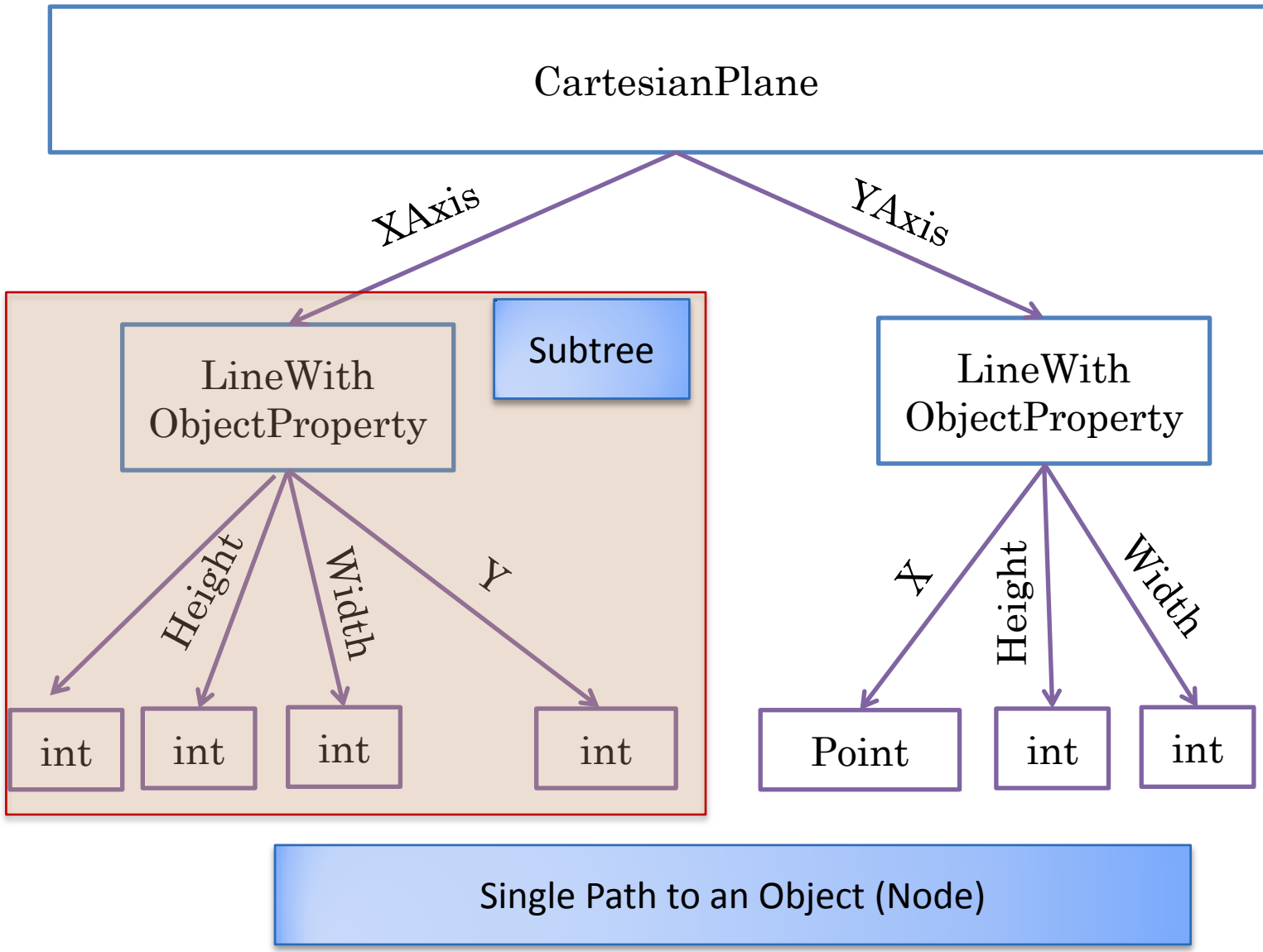
Comp 401
Recursive Tree Traversal and Visitor Design Pattern

Instructor: Prasun Dewan

Prerequisite

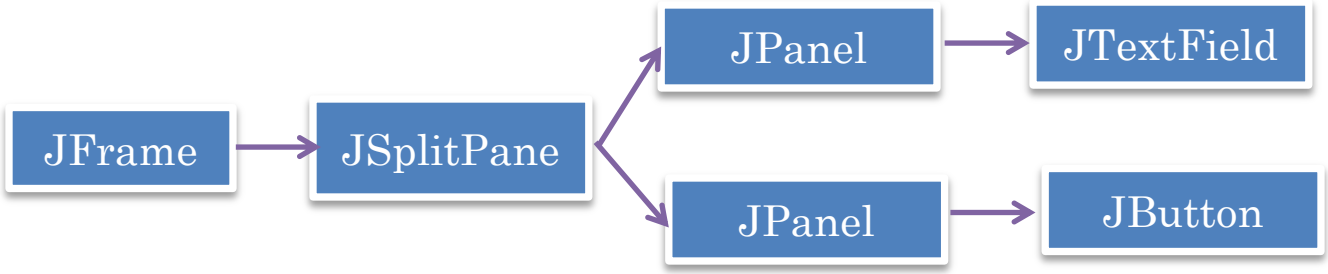
- Recursion
- Composite Design Pattern

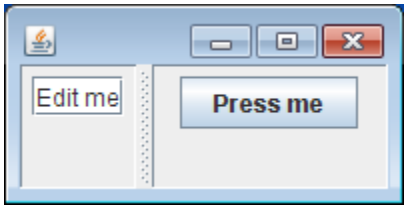
Tree


CartesianPlane
XAxis
YAxis
LineWith ObjectProperty
Subtree
Height
Width
Y
int
int
int
int
LineWith ObjectProperty
X
Height
Width
Point
int
int
Single Path to an Object (Node)

WindowTreeCreator

```
  public static JFrame createTree {
    JFrame frame = new JFrame();
    JSplitPane splitPane = new JSplitPane();
    frame.add(splitPane);
    JPanel leftPanel = new JPanel();
    JPanel rightPanel = new JPanel();
    splitPane.setLeftComponent(leftPanel);
    splitPane.setRightComponent(rightPanel);
    JTextField textField = new JTextField("Edit me");
    leftPanel.add(textField);
    JButton button = new JButton ("Press me");
    rightPanel.add(button);
    frame.setSize(200, 100);
    frame.setVisible(true);
    return frame;
}
```





Window Tree Morpher: Coloring the Tree

```
public static void main(String[] args) {
   JFrame aFrame = WindowTreeCreator.createTree();
   Container root = aFrame.getContentPane();
   colorSubtree(root);
}
```

After a tree has been built, we want to "traverse" it

A la after house has been built we want to color it, put carpets, …

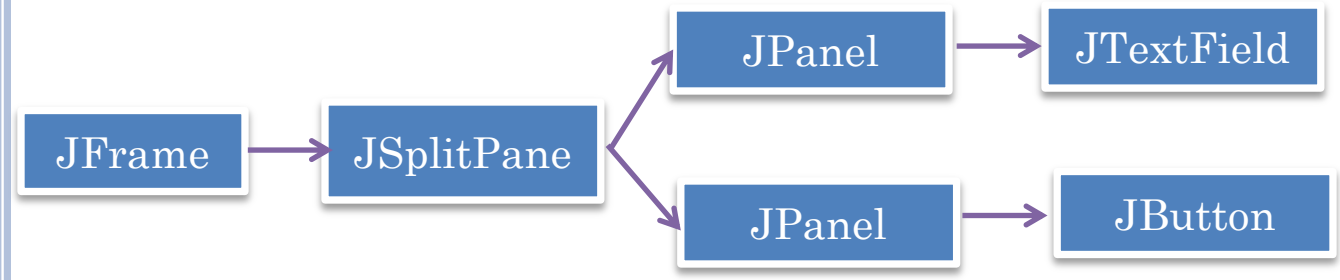
While building we did not do these actions

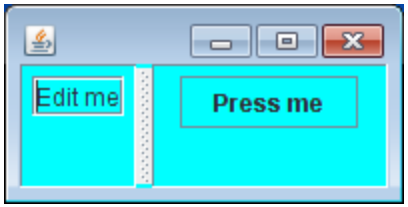
Birds, fertilizers visiting apple tree

Traverse: "visit" or process every node in the tree

Recursion?

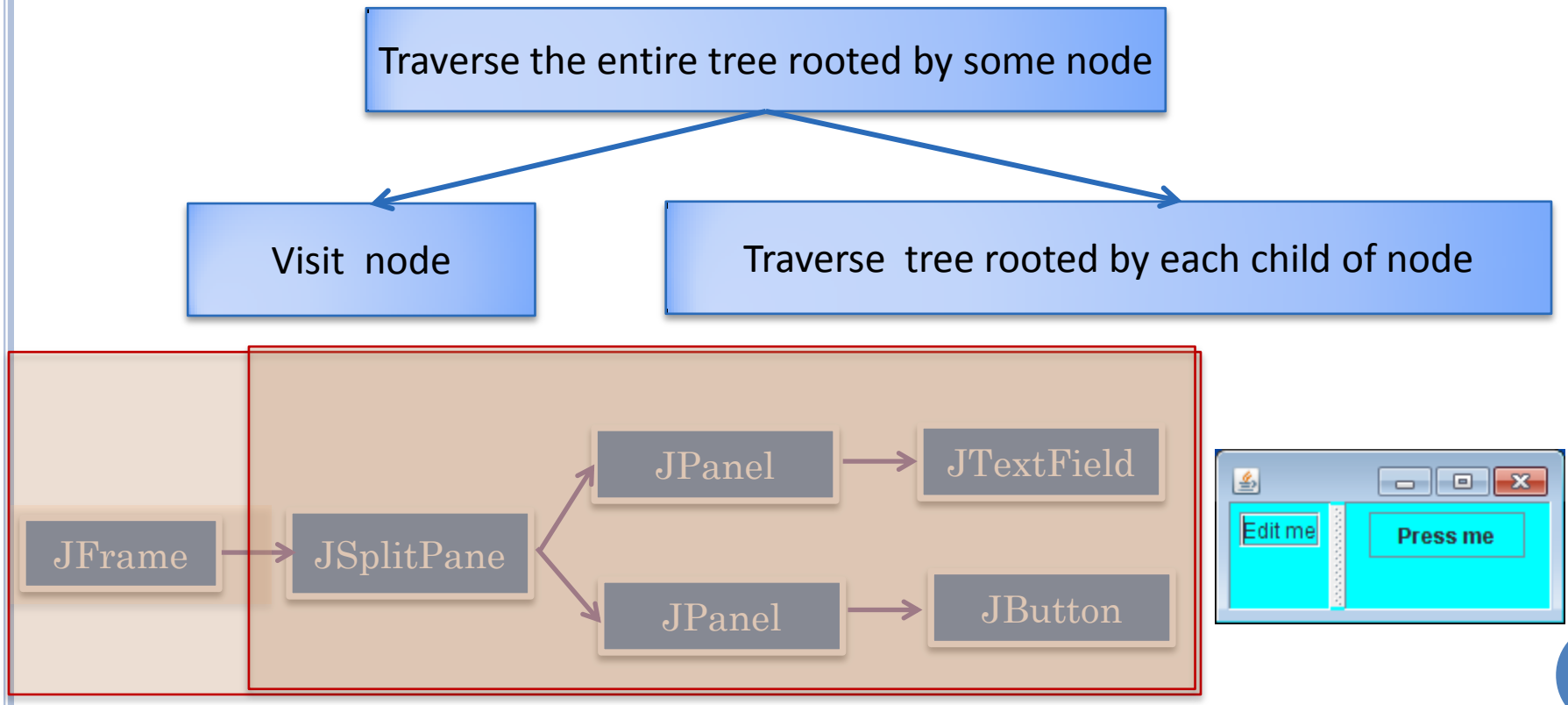
Traverse node by visiting the node and traversing the subtree rooted by each child





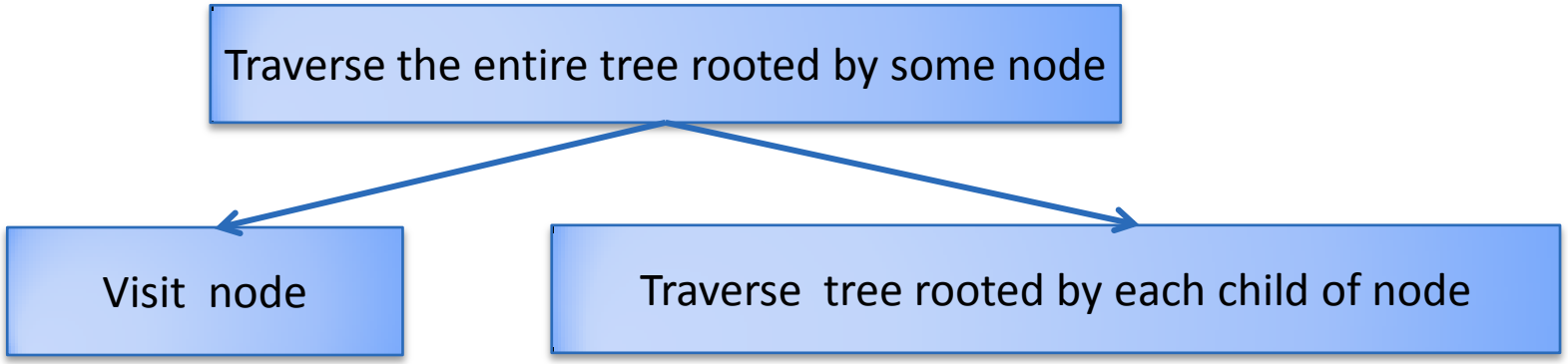
Recursive Traversal

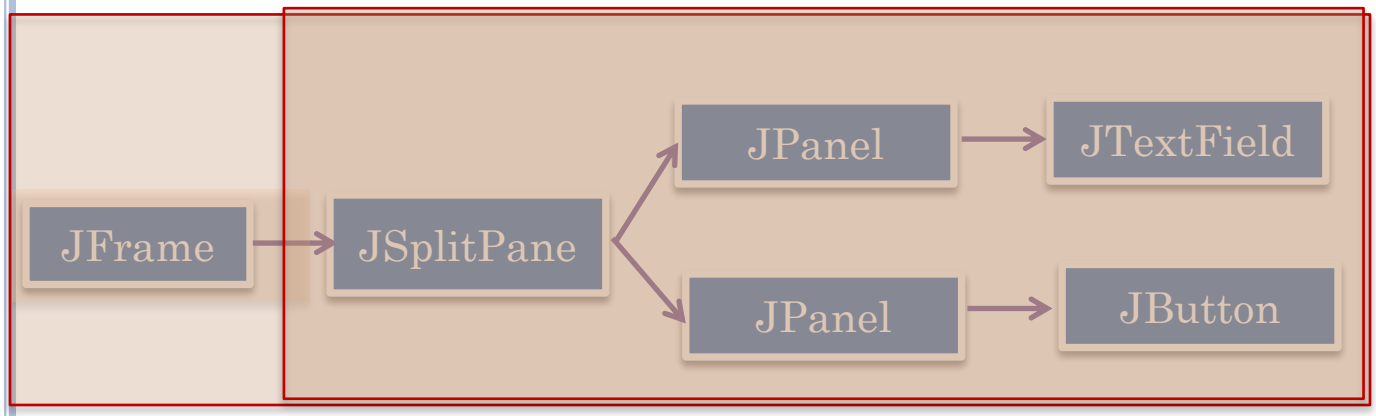
```
public static void main(String[] args) {
   JFrame aFrame = WindowTreeCreator.createTree();
   Container root = aFrame.getContentPane();
   colorSubtree(root);
}
```

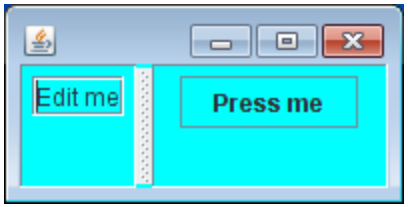


Recursive Traversal?

```
public static void color(Component aComponent) {
   aComponent.setBackground(COLOR);
}
public static void colorSubtree(Component aComponent) {
   …
}
```







Coloring Morpher: Visitor and Traverser Methods

```
public static void color(Component aComponent) {
  aComponent.setBackground(COLOR);
}
public static void colorSubtree(Component aComponent) {
  color(aComponent);
  if (!(aComponent instanceof Container))return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      colorSubtree(components[i]);
    }
  }
}
```

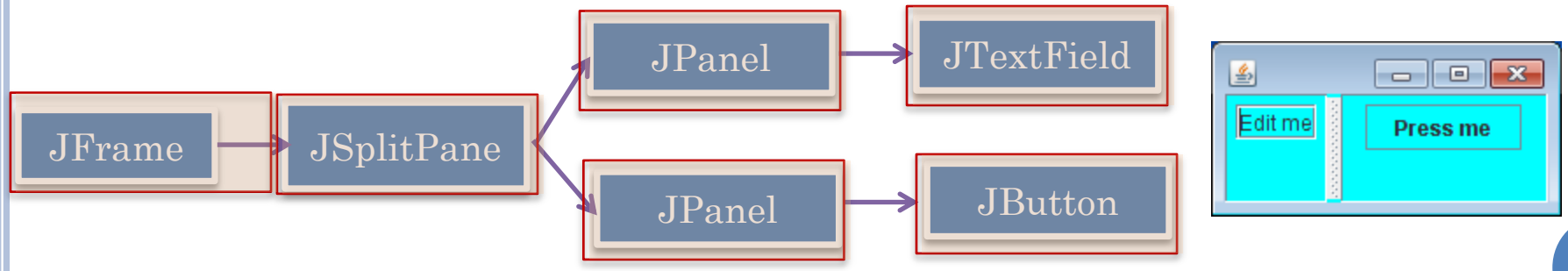
Terminating case

Recursive Reduction Step

Visit Node: Process node

Tree traversal: visit all nodes in tree

Preorder traversal: visit parent before any child



Recursive Methods

- Should have base (terminating) case(s)
- Recurse on smaller problem(s)
  - recursive calls should converge to base case(s)

Recursion Common Properties

Terminating case

Recursive Reduction Steps

```
public static int factorial(int n) {
        if (n == 0)
                return 1;
        else if (n < 0)
                return factorial(-n);
        else
                return n*factorial(n-1);
}
```

```
public static void color(Component aComponent) {
aComponent.setBackground(COLOR);
}
public static void colorSubtree(Component aComponent) {
  color(aComponent);
  if (!(aComponent instanceof Container))return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      colorSubtree(components[i]);
     }
   }
}
```

Terminating case

Recursive Reduction Step

Number-based Recursion

- Method gets some number as argument
- Recursion occurs on smaller or larger number
- Recursion ends when some number argument reaches some limit

```
public static int factorial(int n) {
        if (n == 0)
                return 1;
        else if (n < 0)
                return factorial(-n);
        else
                return n*factorial(n-1);
}
```

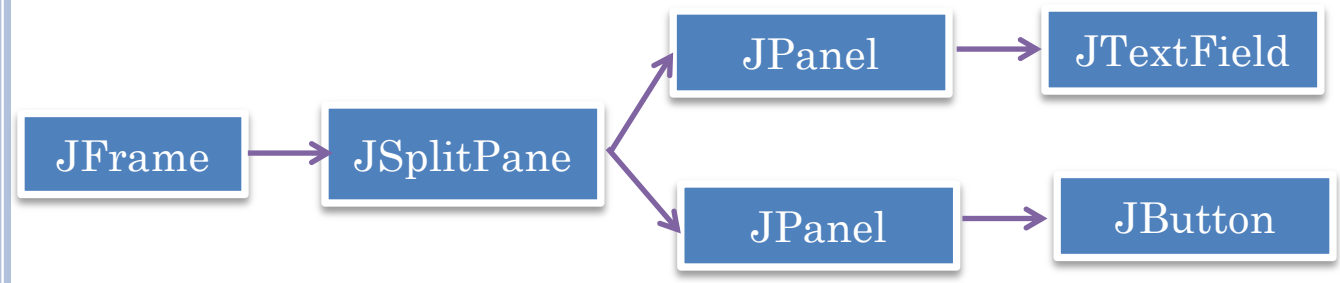
Tree-based Recursion

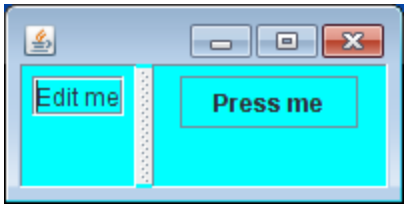
- Method gets some tree item as argument
- Processes or **visits** node.
  - Leaf and non-leaf processing can be different
- Recursion occurs on children
- Recursion ends when leaf node reached

Coloring Morpher

```
public static void color(Component aComponent) {
  aComponent.setBackground(COLOR);
}
public static void colorSubtree(Component aComponent) {
  color(aComponent);
  if (!(aComponent instanceof Container))return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      colorSubtree(components[i]);
    }
  }
}
```

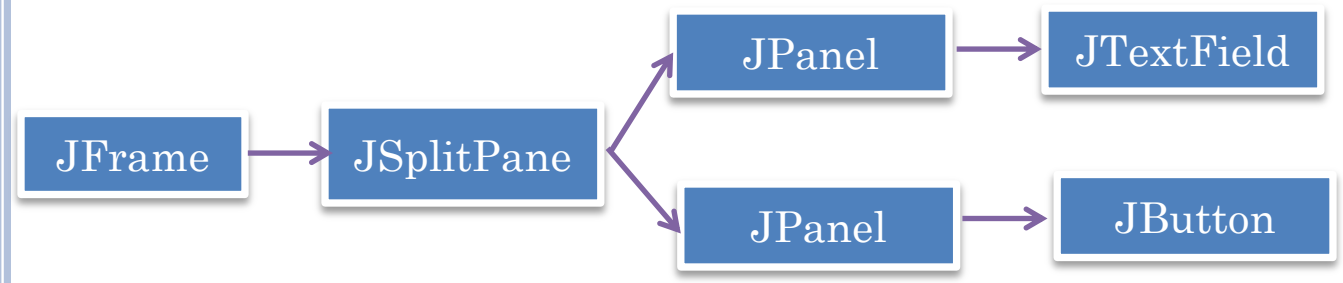
Magnifying morpher?

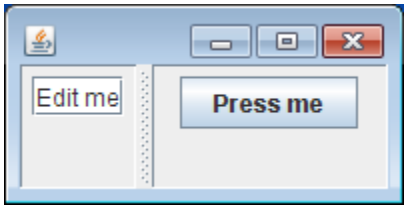




Magnifying Morpher

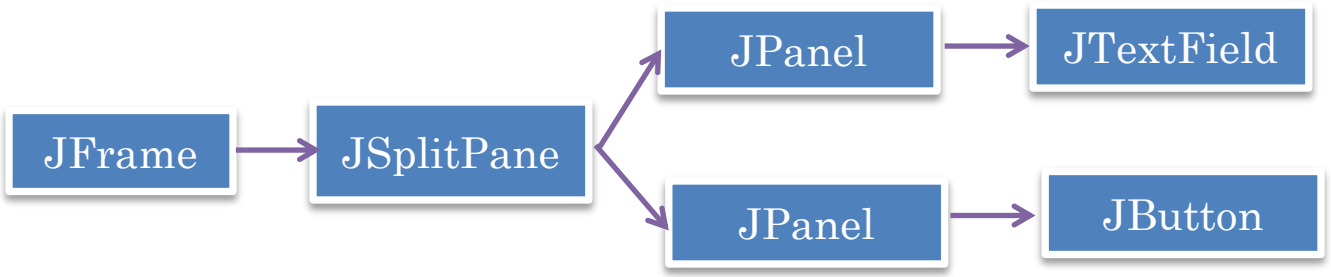
```
public static void magnify(Component aComponent) {
  Dimension aComponentSize = aComponent.getSize();
  aComponent.setSize(
     new Dimension(aComponentSize.width * MAGNIFICATION,
  aComponentSize.height * MAGNIFICATION));
}
public static void magnifySubtree(Component aComponent) {
  magnify(aComponent);
  if (!(aComponent instanceof Container))return;
  Container aContainer = (Container) aComponent;
  Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    magnifySubtree(components[i]);
  }
}
```

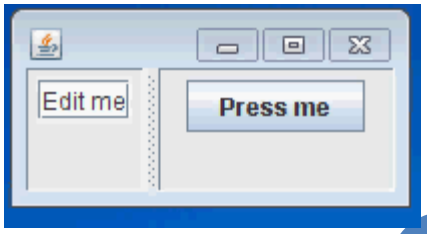




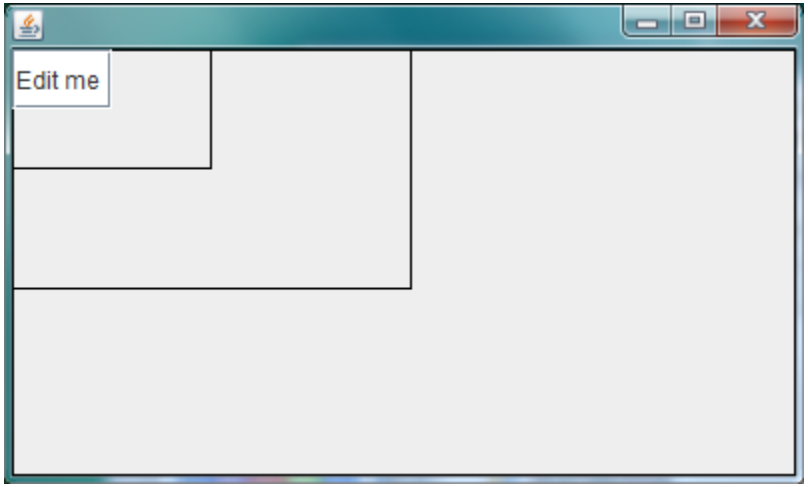
Animated Magnifying Morpher

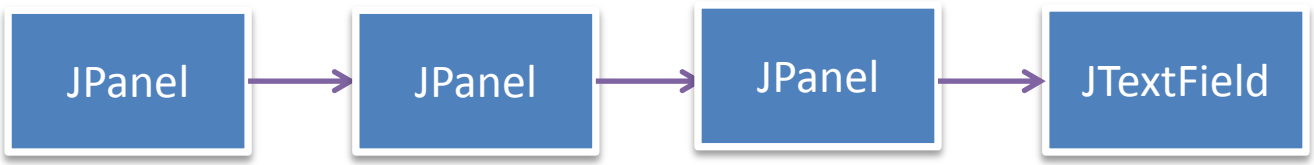
```
public static void animatedMagnify(Component aComponent) {
  magnify(aComponent);
  ThreadSupport.sleep(1000);
}
public static void animatedMagnifySubtree(Component aComponent){
  animatedMagnify(aComponent);
  if (!(aComponent instanceof Container))
    return;
  Container aContainer = (Container) aComponent;
  Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    animatedMagnifySubtree(components[i]);
  }
}
```





Different Tree for Magnification: Java Swing Composite





Swing Composite Creator

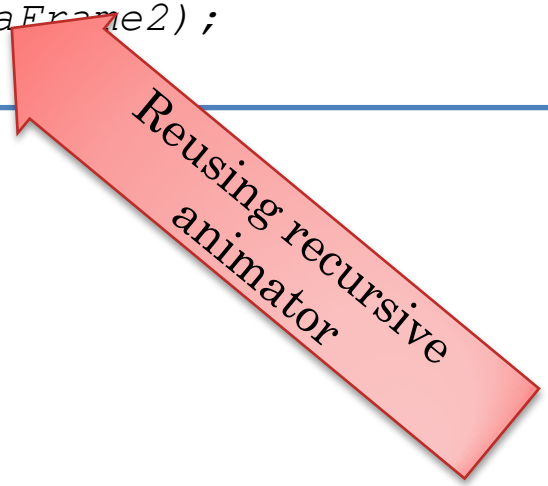
```
public class CompositeSwingComponentCreator {
  …
  public static void main(String[] args) {
    createAndDisplayCompositeUI();
  }
  public static JFrame createAndDisplayCompositeUI () {
    Component leaf = createLeaf(LEAF_SIZE);
    Component root = createComponentTree(leaf);
    JFrame frame = createFrameAndDisplayInWindow(root);
    return frame;
  }
  public static Component createComponentTree(Component leaf) {
    Component retVal = leaf;
    for (int i = 1; i <= NUM_LEVELS; i++) {
      retVal = nestComponent(retVal);
    }
    return retVal;
   }
```

Setting Layout

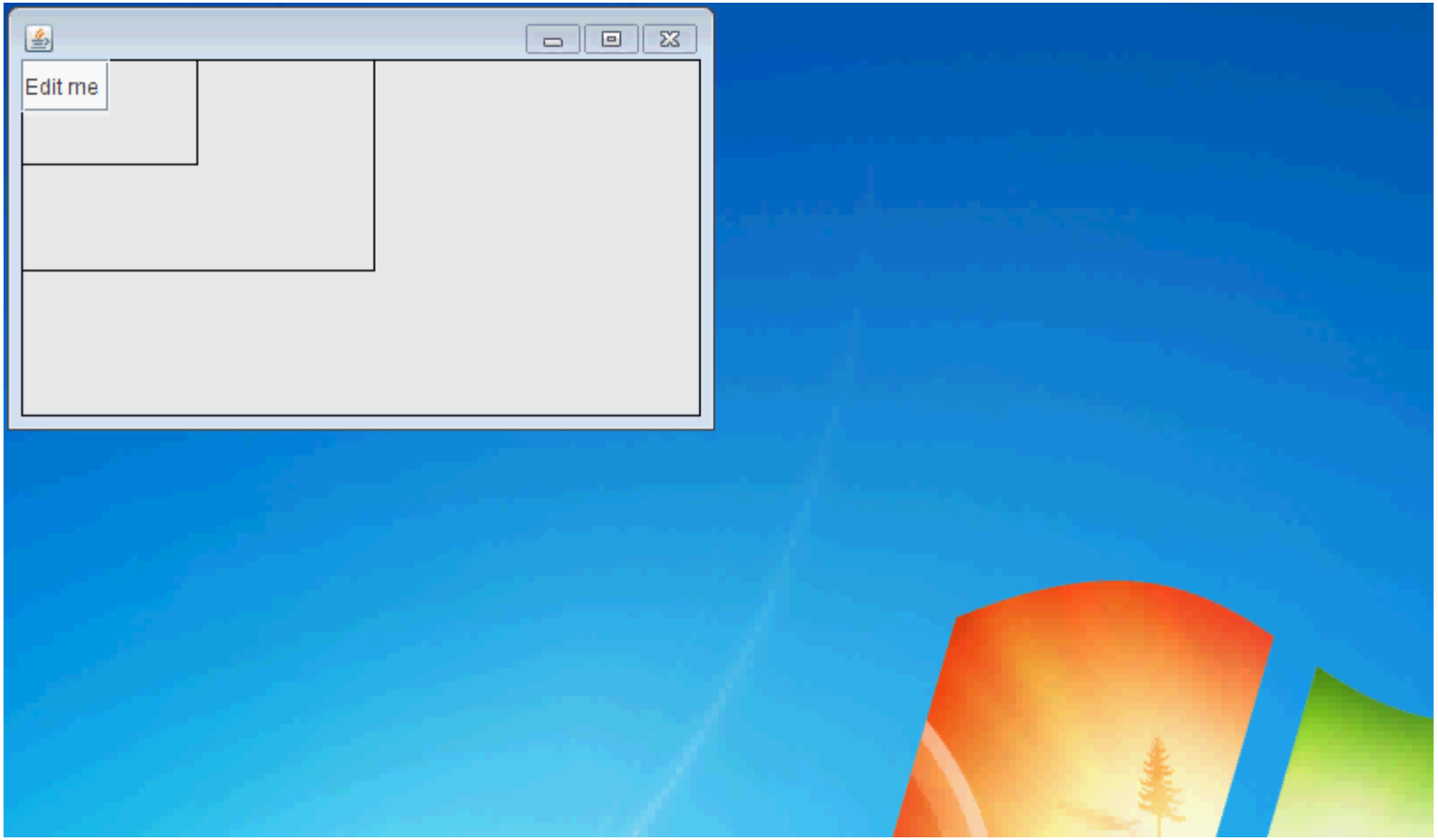
```
public static Component createLeaf(Dimension size) {
  Component retVal = new JTextField("Edit me");
  retVal.setSize(size);
  return retVal;
}
public static Container nestComponent(Component inner) {
   JPanel retVal = new JPanel();
   retVal.setBorder(border); // show outline
   retVal.setSize(
     inner.getWidth() * RELATIVE_SIZE,
     inner.getHeight()* RELATIVE_SIZE);
   retVal.setLayout(null); // do not mess with the size
   retVal.add(inner);
   return retVal;
}
```

Composite Morpher

```
public static void main(String[] args) {
   JFrame aFrame2 = CompositeSwingComponentCreator.
                createAndDisplayCompositeUI();
   Container root2 = aFrame2.getContentPane();
   animatedMagnifySubtree(aFrame2);
}
```

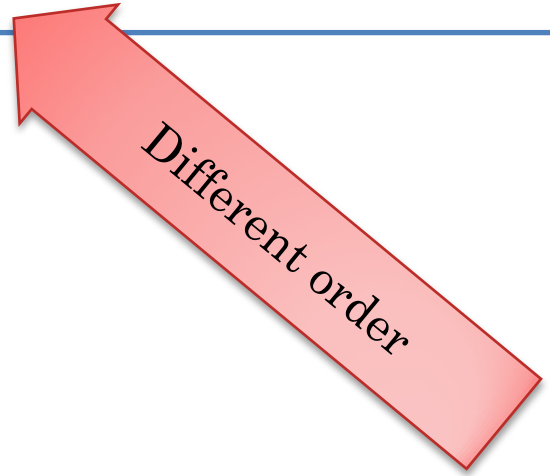


Composite Preorder Morpher

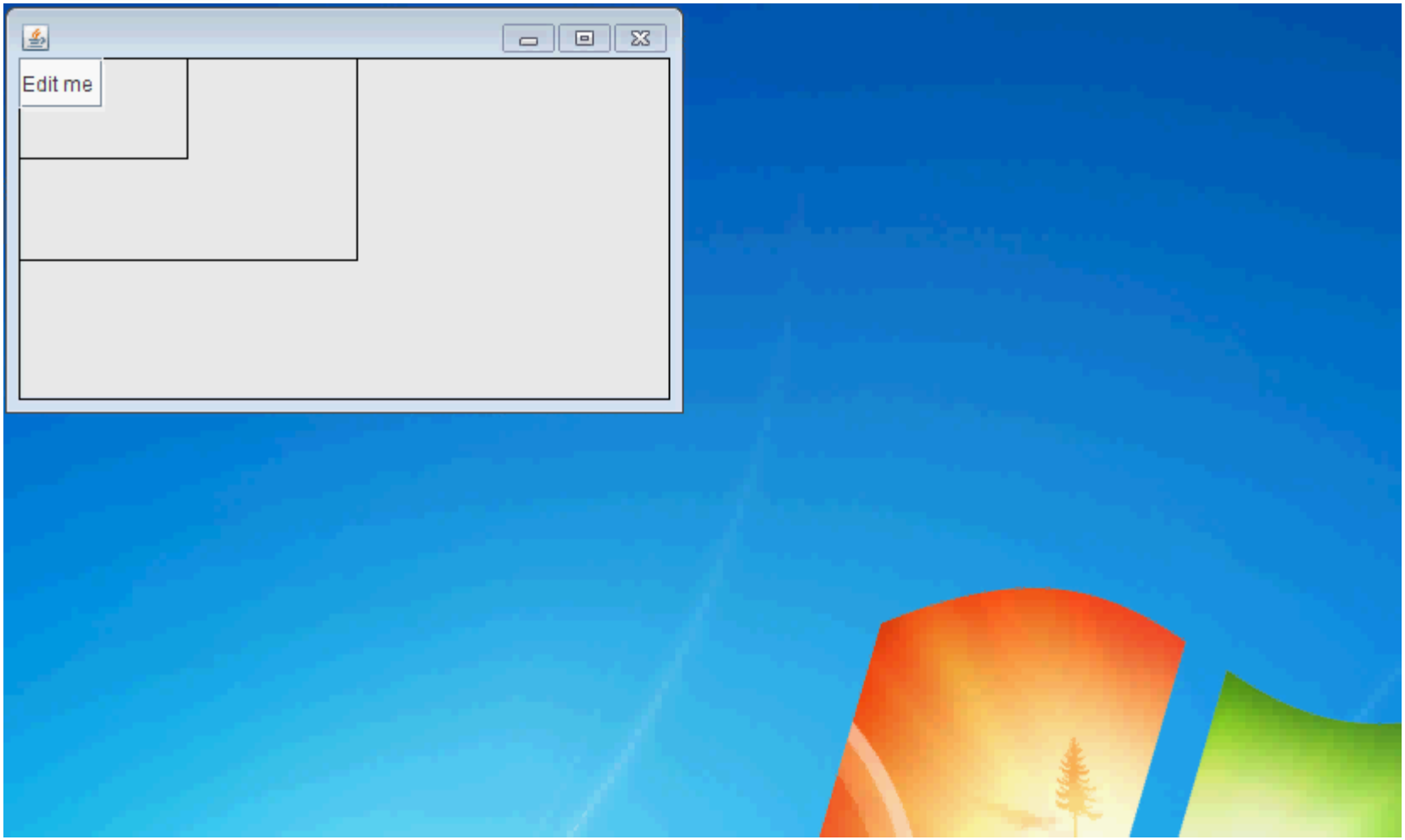


Composite Morpher

```
public static void main(String[] args) {
  JFrame aFrame2 = CompositeSwingComponentCreator.
                createAndDisplayCompositeUI();
  Container root2 = aFrame2.getContentPane();
  animatedMagnifySubtreePostOrder(aFrame2);
}
```



Composite PostOrder Morpher



Previous Preorder Animated Magnifier

```
public static void animatedMagnifySubtree(Component aComponent){
    animatedMagnify(aComponent);
    if (!(aComponent instanceof Container))
      return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      animatedMagnifySubtree(components[i]);
    }
  }
```

Visit node before traversing children

New Postorder Animated Magnifier

```
public static void animatedMagnifySubtreePostOrder(
                                Component aComponent) {
  if (aComponent instanceof Container) {
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      animatedMagnifySubtreePostOrder(components[i]);
    }
  }
  animatedMagnify(aComponent);
}
```

Visit node after traversing children

Tree-based Recursion

- Method gets some tree item as argument
- Processes or **visits** node.
  - Leaf and non-leaf processing can be different
- Recursion occurs on children
- Recursion ends when leaf node reached

Pre-Order and Post-order Tree-based Recursion

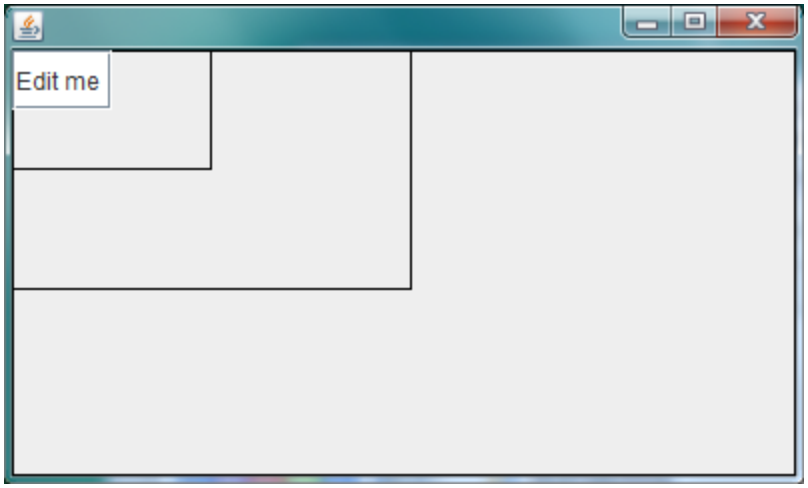
- Preorder: parent node visited before children are visited
- Post-order: parent node visited after children are visited
- In our examples, no difference in final user interface, in general there can be

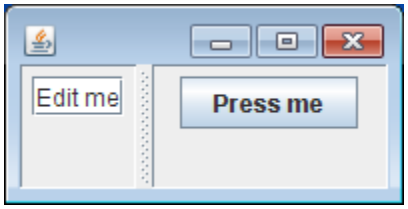
Code-Sharing: Tree Independence

```
public static void animatedMagnifySubtree(Component aComponent){
    animatedMagnify(aComponent);
    if (!(aComponent instanceof Container))
      return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      animatedMagnifySubtree(components[i]);
    }
  }
```

Used the same traverser and visitor for two different trees

Code is tied to Component and Container and not specific subclasses





Separation of Visitor and Traverser Method

```
public static void animatedMagnify(Component aComponent) {
  magnify(aComponent);
  ThreadSupport.sleep(1000
}
```

Traverser method

Visitor method def

```
 static void animatedMagnifySubtree(Component aComponent){
  animatedMagnify(aComponent);
  if (!(aComponent instanceof Container))
    return;
  Container aContainer = (Container) aComponent;
  Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    animatedMagnifySubtree(components[i]);
  }
}
```

Visitor method call

Separation benefits?

Sharing the Visitor Method

```
public static void animatedMagnify(Component aComponent) {
  magnify(aComponent);
  ThreadSupport.sleep(1000
}
```

Different Traverser method

Same visitor method def

```
ic static void animatedMagnifySubtreePostOrder(
                              Component aComponent) {
  if (aComponent instanceof Container) {
  Container aContainer = (Container) aComponent;
  Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    animatedMagnifySubtreePostOrder(components[i]);
  }
}
  animatedMagnify(aComponent);
}
```

Same visitor method call

Separation of Concerns

Make two pieces of code referring to each other different if one can be used without the other

We might want to keep the visitor code constant and vary the traverser code

Do not have to re-implement the constant code visitor for each kind of traverser

We might want to keep the traverser code constant and vary the visitor code

Do we have to re-implement the constant code in traverser for each kind of visitor?

Repeating Constant Code

```
public static void magnifySubtree(Component aComponent) {
    magnify(aComponent);
    if (!(aComponent instanceof Container))return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      magnifySubtree(components[i]);
    }
  }
```

All other code remains constant

```
public static void colorSubtree(Component aComponent) {
  color(aComponent);
  if (!(aComponent instanceof Container))return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      colorSubtree(components[i]);
    }
  }
}
```

Do we have to re-implement the constant code in traverser for each kind of visitor?

Switch approach: Specifying Visitor

```
public enum VisitOption {
        MAGNIFY,
        ANIMATED_MAGNIFY,
        COLOR
}
```

Visitor Enum Parameter

```
public static void traverseInOrder(Component aComponent,
                                   VisitOption aVisitOption) {
  switch (aVisitOption) {
    case MAGNIFY:
      magnify(aComponent);
      break;
   case ANIMATED_MAGNIFY:
      animatedMagnify(aComponent);
      break;
   case COLOR:
      color(aComponent);
      break;
  }
  if (!(aComponent instanceof Container))
    return;
  Container aContainer = (Container) aComponent;
  Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    traverseInOrder(components[i], aVisitOption);
  }
}
```

Specifying Visitor Enum Option

```
public static void main(String[] args) {
  JFrame aFrame = WindowTreeCreator.createTree();
  Container root = aFrame.getContentPane();
  traverseInOrder(root, VisitOption.COLOR);
}
```

Problem?

```
public static void traverseInOrder(Component aComponent,
                                   VisitOption aVisitOption) {
  switch (aVisitOption) {
    case MAGNIFY:
      magnify(aComponent);
      break;
   case ANIMATED_MAGNIFY:
      animatedMagnify(aComponent);
      break;
   case COLOR:
      color(aComponent);
      break;
  }
  if (!(aComponent instanceof Container))
    return;
  Container aContainer = (Container) aComponent;
  Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    traverseInOrder(components[i], aVisitOption);
  }
}
```

Must change traverser or write new one for unanticipated visitor

Action vs. Command vs. Visitor Object

Action Object (Method Object)	execute (targetObject, params)
Commmand (action with target + parameters)	execute () Constructor (targetObject, params)
Visitor (action with parameters)	execute (targetObject) Constructor (params)

Need an object that allows the same method to be executed on different nodes of a tree

Need an object that adds parameters to an action object and subtracts target object from command

Execute in (a) command takes no params, (b) action takes target object and params, (c) visitor takes target object

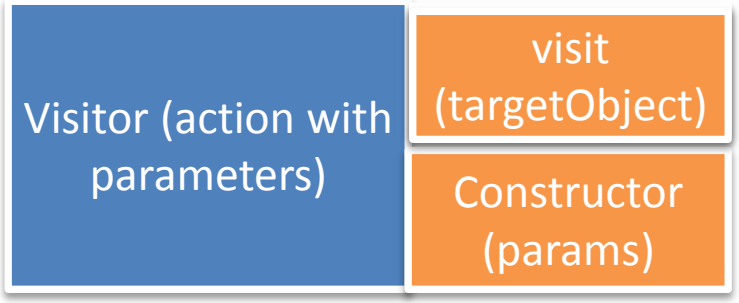
Constructor in (a) command takes target and params, (b) action takes no params, and (c) visitor takes params as arguments.

Visitor Object

Provides a execute operation to perform some embedded operation.

The execute operation takes target object as argument

Called visit



Component Visitor

```
public interface ComponentVisitor {
  public void visit(Component aComponent);
}
```

```
public class AComponentMagnifier implements ComponentVisitor {
  int magnification;
  public AComponentMagnifier (int aMagnification) {
    magnification = aMagnification;
  }
  public void visit(Component aComponent) {
    Dimension aComponentSize = aComponent.getSize();
    aComponent.setSize(aComponentSize.width*magnification,
    aComponentSize.height*magnification);
  }
}
```

Component Visitor

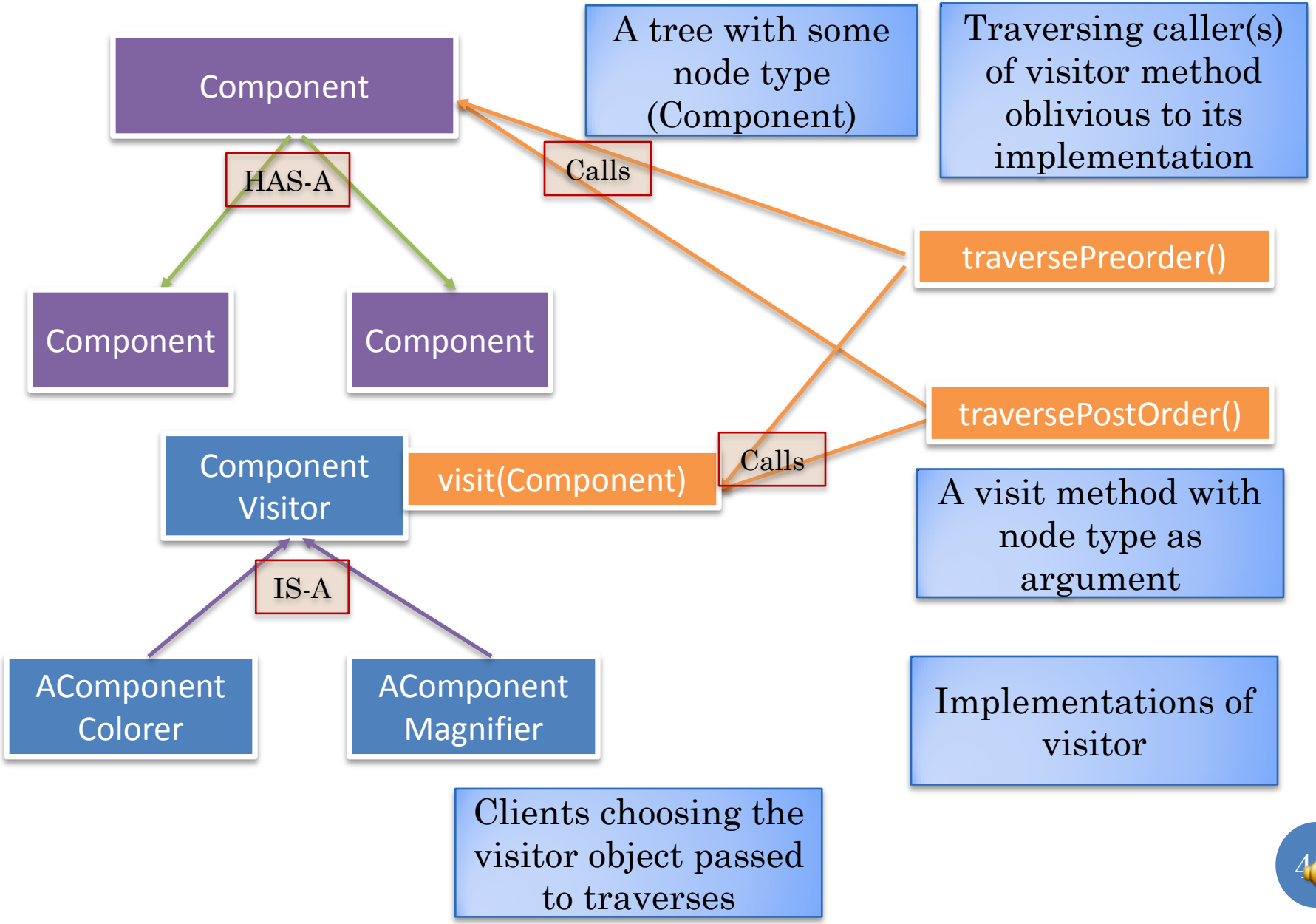
```
public interface ComponentVisitor {
  public void visit(Component aComponent);
}
```

```
public class AComponentColorer implements ComponentVisitor {
  Color color;
  public AComponentColorer(Color aColor) {
    color = aColor;
  }
  public void visit(Component aComponent) {
    aComponent.setBackground(color);
  }
}
```

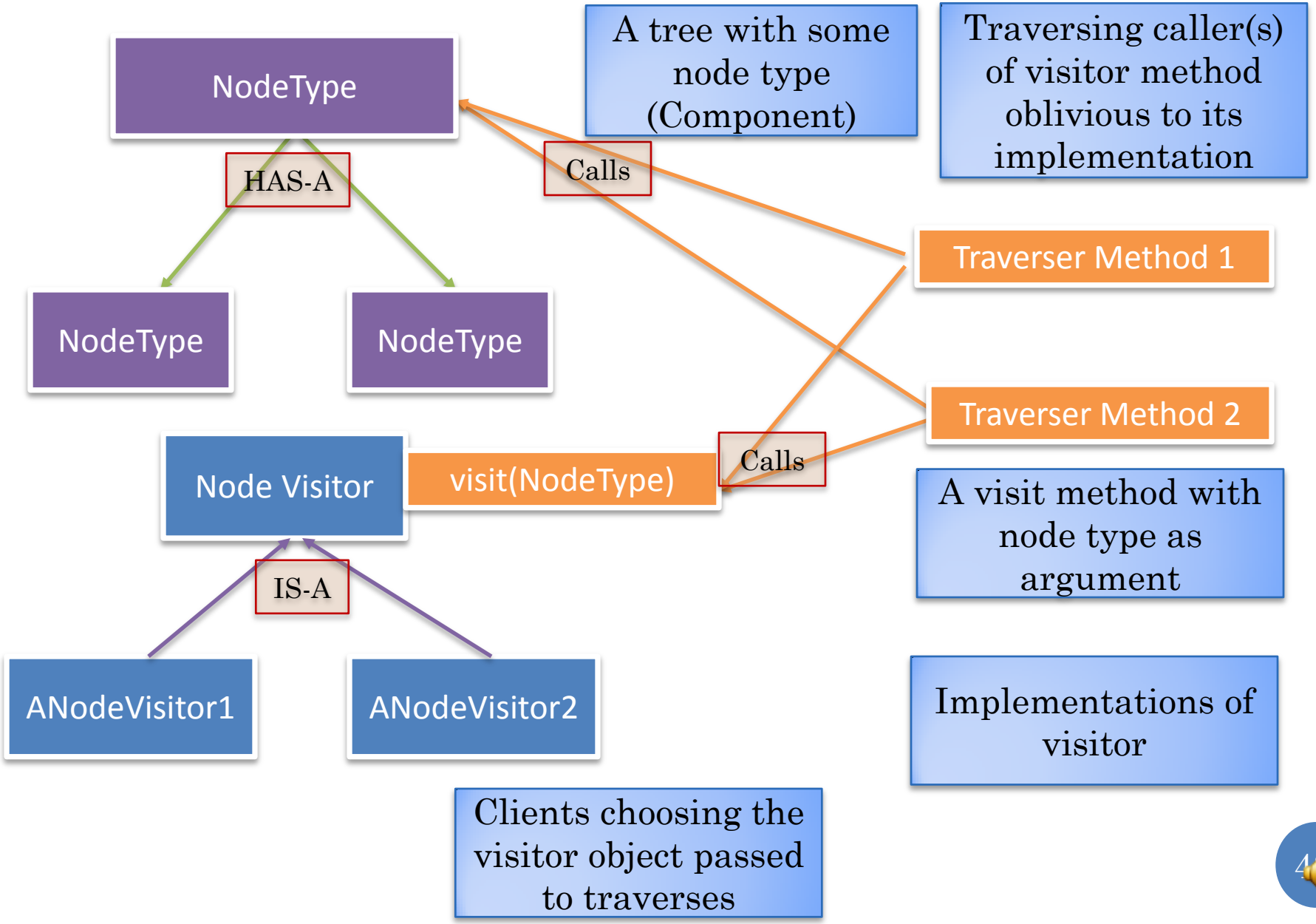
Visitor as Method Parameter

```
public static void traversePreOrder(Component aComponent,
                                    ComponentVisitor aVisitor) {
  aVisitor.visit(aComponent);
  if (!(aComponent instanceof Container)) return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    traversePreOrder(components[i], aVisitor);
  }
}
```

Example of Visitor Pattern



Abstract Visitor Pattern



Everyday Visitor Objects

- Compiler visitors for:
  - Formatting program elements
  - Refactoring program elements
  - Compiling program elements.
- ObjectEditor visitors for:
  - Attaching widgets to object components.
  - Registering listeners of object components.
  - Printing a textual representation of object components.

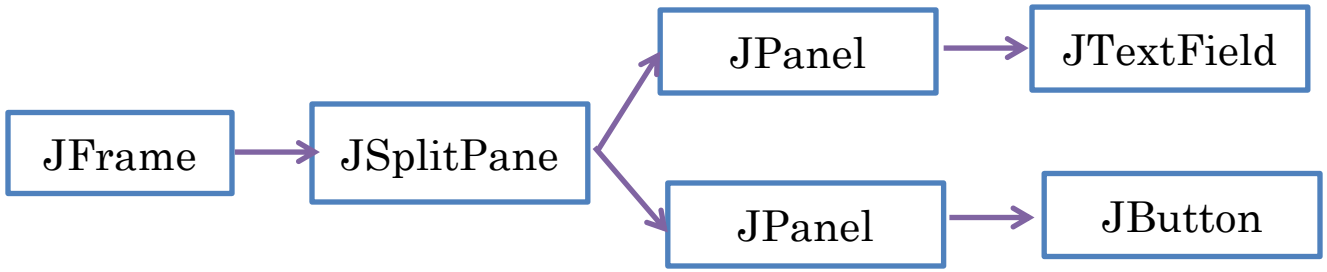
Visitor Pattern

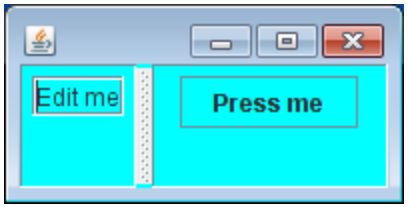
- Some composite tree with noted of type <T>
- Visitor interface with visitor method
  - **public interface** <VisitorInterface> {**public** <SomeType> <VisitMethodName> (<T> <p>);}
- One or more traverser methods that use Composite and Visitor interface to pass one or more tree nodes to the visitor method.
  - <SomeType> <TraverserName> (<T> <nodeParam>, <VisitorInterface> <visitorParam>
- Implementations of visitor interface whose constructors take as arguments external variables that need to be accessed by the visitor method
  - **public class** <VisitorClass> **implements** <VisitorInterface> {**public** <VisitorClass> (<T1> <p1>, <T2> <p2>, … <Tn> <pn>) { … }, … }
- Some client of the pattern calls the traverser with an instance of one of the implementations:
  - <TraverserName> (<T Value>, <Visitor Interface Value>)

Extra

Coloring the Components

```
  public static JFrame createTree {
    JFrame frame = new JFrame();
    JSplitPane splitPane = new JSplitPane();
    frame.add(splitPane);
    JPanel leftPanel = new JPanel();
    JPanel rightPanel = new JPanel();
    splitPane.setLeftComponent(leftPanel);
    splitPane.setRightComponent(rightPanel);
    JTextField textField = new JTextField("Edit me");
    leftPanel.add(textField);
    JButton button = new JButton ("Press me");
    rightPanel.add(button);
    frame.setSize(200, 100);
    frame.setVisible(true);
    return frame;
}
```



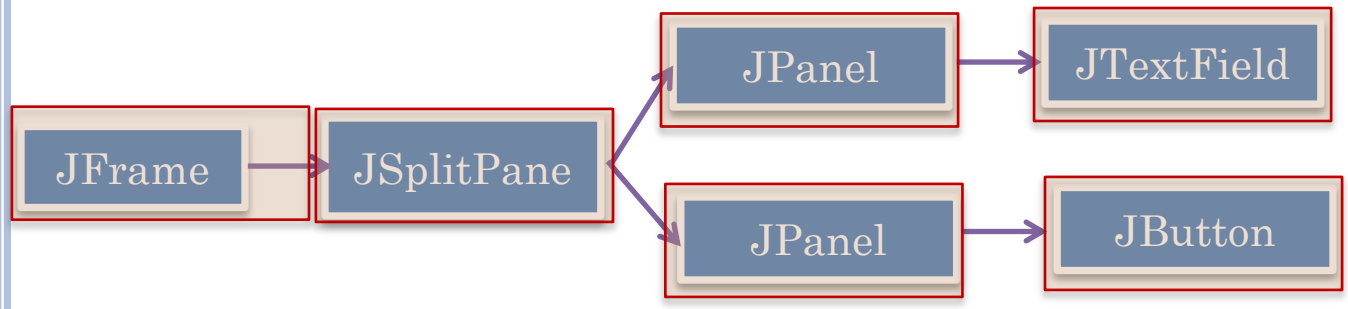


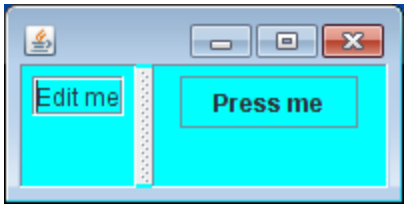
Coloring Morpher: Visitor and Traverser Methods

```
public static void color(Component aComponent) {
  aComponent.setBackground(COLOR);
}
public static void colorSubtree(Component aComponent) {
  color(aComponent);
  if (!(aComponent instanceof Container))return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
    for (int i = 0; i < components.length; i++) {
      colorSubtree(components[i]);
     }
   }
}
```

Terminating case

Recursive Reduction Step





Specifying Visitor Method and Target

```
public static void main(String[] args) {
  JFrame aFrame = WindowTreeCreator.createTree();
  Container root = aFrame.getContentPane();
  try {
    Method colorMethod = SwingComponentTreeMorpher.class.
        getDeclaredMethod("color", new Class[]{Component.class});
    traverseInOrder(root, colorMethod,
        SwingComponentTreeMorpher.class);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

```
public class SwingComponentTreeMorpher {
  …
  public static void color(Component aComponent) {
    aComponent.setBackground(COLOR);
  }
}
```

Better Language Support

```
public static void main(String[] args) {
  JFrame aFrame = WindowTreeCreator.createTree();
  Container root = aFrame.getContentPane();
  try {
    Method colorMethod = SwingComponentTreeMorpher.class.
        SwingComponentTreeMorpher.color());
    traverseInOrder(root, colorMethod,
        SwingComponentTreeMorpher.class);
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

```
public class SwingComponentTreeMorpher {
  …
  public static void color(Component aComponent) {
    aComponent.setBackground(COLOR);
  }
}
```

Visitor Method (and Object) Parameter

```
public static void traverseInOrder(Component aComponent,
        Method aVisitorMethod, Object aVisitorMethodTarget) {
  try {
    aVisitorMethod.invoke(aVisitorMethodTarget,
                          new Component[] {aComponent});
  } catch (Exception e) {
    e.printStackTrace();
  }
  if (!(aComponent instanceof Container))
    return;
  Container aContainer = (Container) aComponent;
  Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    traverseInOrder(components[i],
        aVisitorMethod,  aVisitorMethodTarget);
  }
}
```

At run time, errors can occur with method being bound to the wrong parameters

No compile time check ensuring correct parameters are bound to method

Method: ActionObject

Action Object = Embedded Opertation

execute (targetObject, params)

At run time, errors can occur with method being bound to the wrong parameters

No compile time check ensuring correct parameters are bound to method

Command Object

Command Object = Embedded Operation + Target + Parameters

execute ()

Constructor (targetObject, params)

Provides a execute operation to perform some embedded operation.

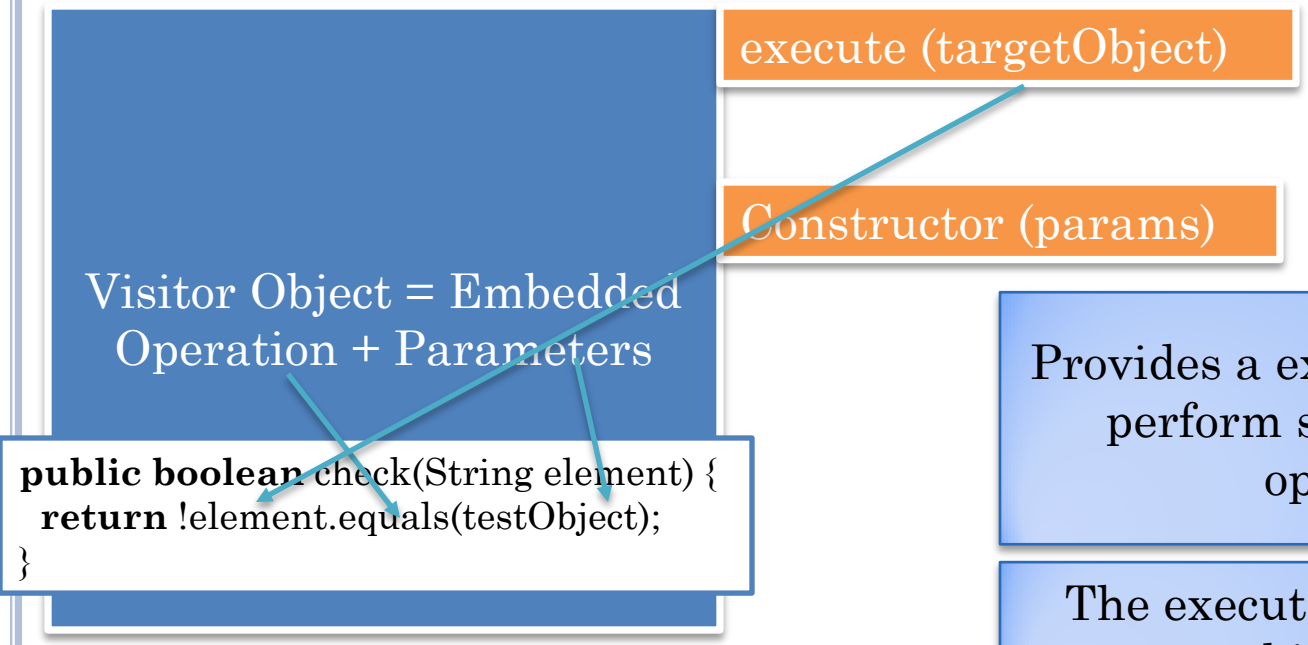
The execute operation takes no arguments.

Constructor takes parameters of operation as arguments.

Action is an operation that can be invoked on many different arguments

A command is a specific action invocation.

Visitor Object



Provides a execute operation to perform some embedded operation.

The execute operation takes target object as argument

Constructor in (a) command takes target and params, (b) action takes no params, and (c) visitor takes params as arguments.

Execute in (a) command takes no params, (b) action takes target object and params, (c) visitor takes target object

Visitor Object

Visitor Object = Embedded Operation + Parameters

visit(targetObject)

Constructor (params)

Visitor Method (and Object) Parameter

```
public static void traverseInOrder(Component aComponent,
        Method aVisitorMethod, Object aVisitorMethodTarget) {
  try {
    aVisitorMethod.invoke(aVisitorMethodTarget,
                          new Component[] {aComponent});
  } catch (Exception e) {
    e.printStackTrace();
  }
  if (!(aComponent instanceof Container))
    return;
  Container aContainer = (Container) aComponent;
  Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    traverseInOrder(components[i],
        aVisitorMethod,  aVisitorMethodTarget);
  }
}
```

Visitor Object Parameter

```
public static void traversePreOrder(Component aComponent,
                                    ComponentVisitor aVisitor) {
  aVisitor.visit(aComponent);
  ThreadSupport.sleep(1000);
  if (!(aComponent instanceof Container)) return;
    Container aContainer = (Container) aComponent;
    Component[] components = aContainer.getComponents();
  for (int i = 0; i < components.length; i++) {
    traversePreOrder(components[i], aVisitor);
  }
}
```

Visitor Method (and Object) Parameter

```
public static void traverseInOrder(Component aComponent,
        Method aVisitorMethod, Object aVisitorMethodTarget) {
   try {
     aVisit    hod.invoke(aVisitorMethodTarget,
                          new Component[] {aComponent});
   } c      ption e) {
          ackTrace();

     aComponent instanceof Containe
     eturn;
   Container aContainer = (Containe      ent;
   Component[] components = aContaine      omponents();
   for (int i = 0; i < components.leng    ; i++) {
     traverseInOrder(components[i],
         aVisitorMethod,  aVisitorMethodTarget);
   }
}
```

Invoke aVisitorMethod on aComponent

Component is the real target

Possible to pass to method invoke the wrong target object and parameters

Action vs. Command vs. Visitor Object

Action Object (Method Object)	execute (targetObject, params)

Commmand (action with target + parameters)	execute ()
	Constructor (targetObject, params)

Need an object that adds parameters to an action object and subtracts target object from command

Execute in (a) command takes no params, (b) action takes target object and params, (c) visitor takes target object

Visitor (action with parameters)	execute (targetObject)
	Constructor (params)

```
public static void traverseInOrder(Co
        Me                hod, Object
    …
    traverseInOrder(components[i],
       aVisitorMethod,  aVisitorMetho
    …
}
```

Target object of method varies

Constructor in (a) command takes target and params, (b) action takes no params, and (c) visitor takes params as arguments.