Software Development with CaesarJ

CaesarJ Team

Outline

- **Aspects in CaesarJ**
  - **Aspects as classes**
  - **Dynamic aspect deployment**
- Hierarchical Refinements
  - Extending component with virtual classes
  - Combining different extensions
  - Feature-oriented decomposition
- Crosscutting Integration
  - Defining wrapper classes
  - Observing events with pointcuts
  - Variability management
- Integrating Distributed Components

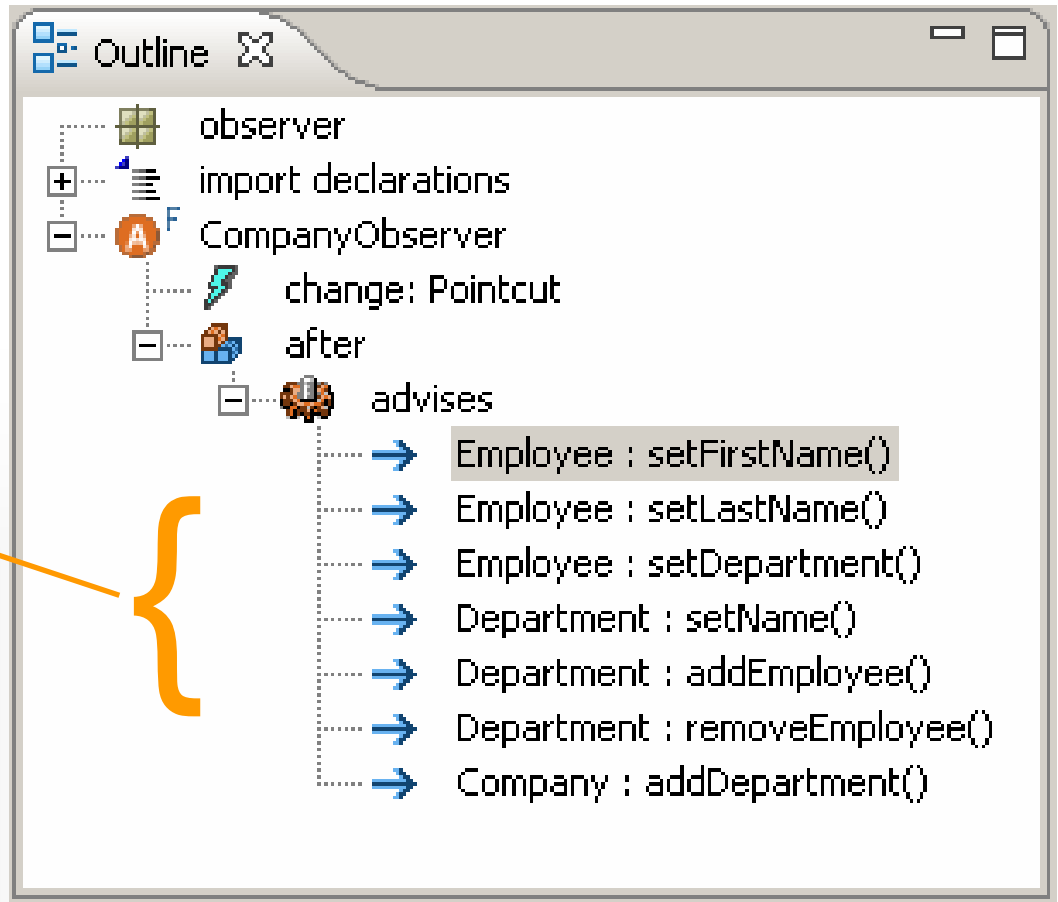
To Do: Getting Started

- Start CJDT v0.8.5
  - start **`CaesarJ 0.8.5`** from your desktop
  - select the training workspace
  - Check that all labs are available (**`LabA`** – **`LabG`**)
- Open **`LabA`**
- Explore the model defined in the company package
  - **`Company`**, **`Department`**, and **`Employee`**
- Explore **`aspects.CompanyChangeTracer`**
  - what is the meaning of the pointcut **`change()`** ?
  - what is the meaning of the advice **`after(): change() {…}`** ?
  - how are primitive pointcuts composed (&&, ||) ?
- Explore **`tests.DriverA`**
- Run **`tests.DriverA`**
  - What behavior was added by the **`CompanyChangeTracer`**?

To Do: Using Outline Views

- Open **`aspects.CompanyChangeTracer`** again
- Explore the crosscutting information in the outline view

Shows which methods are advised by the `CompanyChangeTracer`



Aspects in CaesarJ

```
public deployed cclass CompanyChangeTracer {

   pointcut change() :
               execution(void company.*.set*(..))
            || execution(void company.*.add*(..))
            || execution(void company.*.remove*(..))
            || execution(void Company.transferTo(..));

   after(Object o): change() && this(o) {
      System.out.println(
         "Object '" + o.toString() + "' changed");
   }
}
```

- Aspects in CaesarJ are classes that have pointcuts and pieces of advice
- Keyword **`deployed`** creates and activates a singleton instance of the class

Dynamic Aspect Deployment

But aspects can also be deployed dynamically

```
CompanyChangeTracer asp1 = new CompanyChangeTracer();
CompanyChangeTracer asp2 = new CompanyChangeTracer();

dept.setName("DeptA"); /* not intercepted */
deploy asp1;
dept.setName("DeptB"); /* intercepted by asp1 */
deploy asp2;
dept.setName("DeptC"); /* intercepted by asp1 and asp2 */
undeploy asp1;
undeploy asp2;
dept.setName("DeptD"); /* not intercepted */
```

To Do: Manipulate Aspects

Task:

Trace only the changes to model after loading it from the database. Additionally trace to a log file.

Steps:

- Open **`LabB`**
- Open **`tests.DriverA`**
- Change deployment scope of **`tracer`** so that it observes only changes after loading from the database
- Create and deploy a new instance of **`CompanyChangeTracer`**, which traces company model changes to a log file

Aspect Scoping

Thread local deployment:

```
CompanyChangeTracer asp1 = new CompanyChangeTracer();
deploy(asp1) {
   dept.setName(“DeptA”);
}
```

Object local deployment:

```
CompanyChangeTracer asp1 = new CompanyChangeTracer();
CompanyChangeTracer asp1 = new CompanyChangeTracer();
Department dept1 = new Department(“deptA”);
Department dept2 = new Department(“deptB”);
DeploySupport.deployOnObject(asp1, dept1);
DeploySupport.deployOnObject(asp2, dept2);
dept1.setName(“deptC”);       /* intercepted by asp1 */
dept2.setName(“deptD”);       /* intercepted by asp2 */
dept3.setName(“deptF”);       /* not intercepted */
DeploySupport.undeployFromObject(asp1, dept1);
DeploySupport.undeployFromObject(asp2, dept2);
```

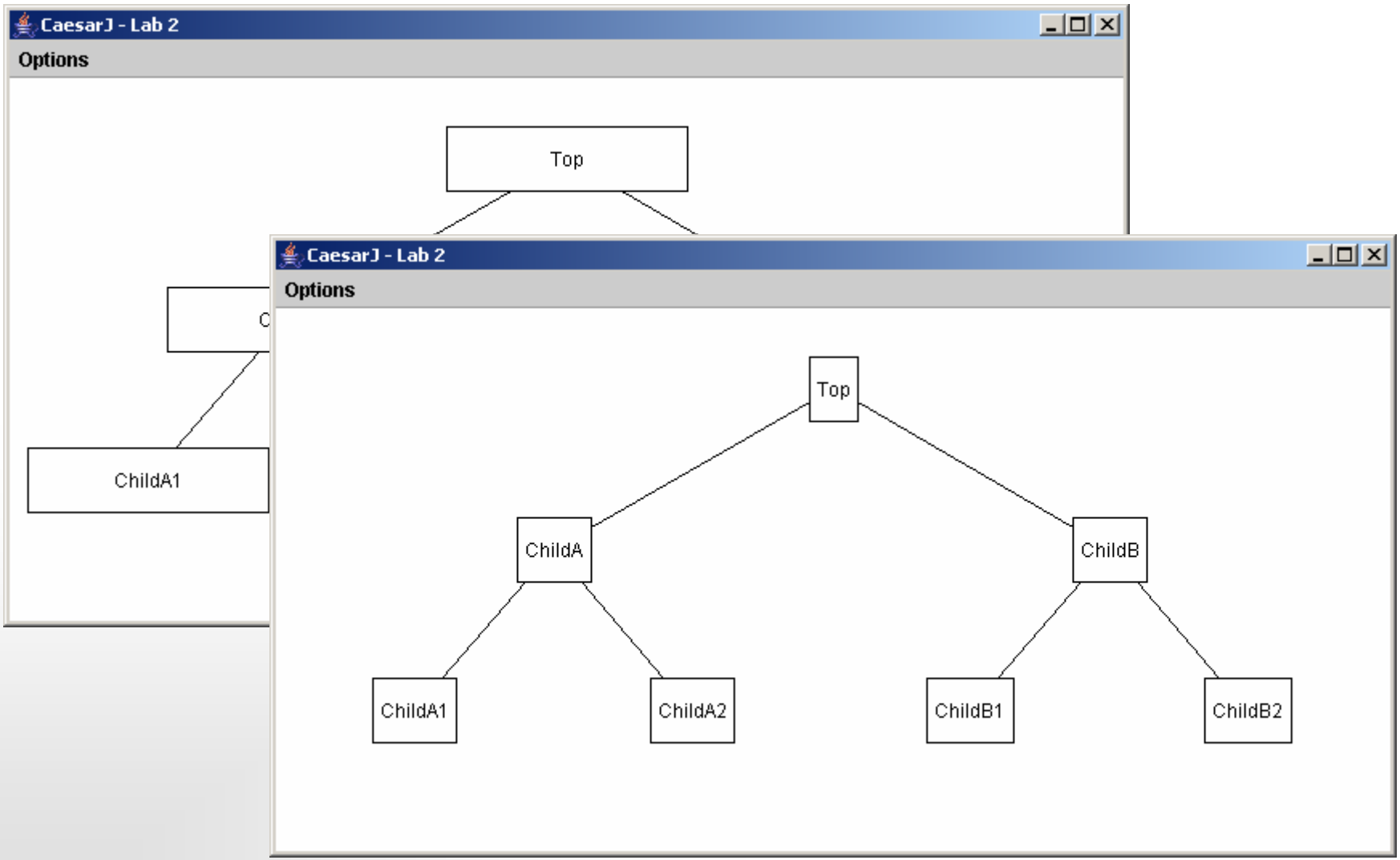
Advantages of Aspects as Objects

- Aspects can be instantiated and referenced as any other object
- References to aspects can be polymorphic
  - This enables a form of aspectual polymorphism
- Multiple instances of an aspect can be created and used simultaneously
- Aspects can contain state
  - To parameterize their behavior
  - To accumulate results of observation
- Aspects can be deployed on different scopes

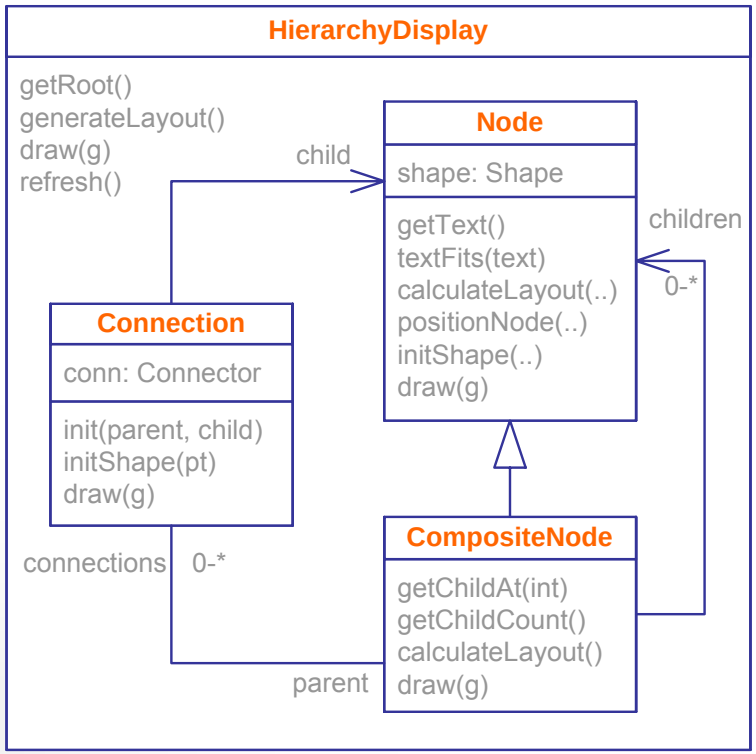
Outline

- Aspects in CaesarJ
  - Aspects as classes
  - Dynamic aspect deployment
- **Hierarchical Refinements**
  - **Extending component with virtual classes**
  - **Combining different extensions**
  - **Feature-oriented decomposition**
- Crosscutting Integration
  - Defining wrapper classes
  - Observing events with pointcuts
  - Variability management
- Integrating Distributed Components

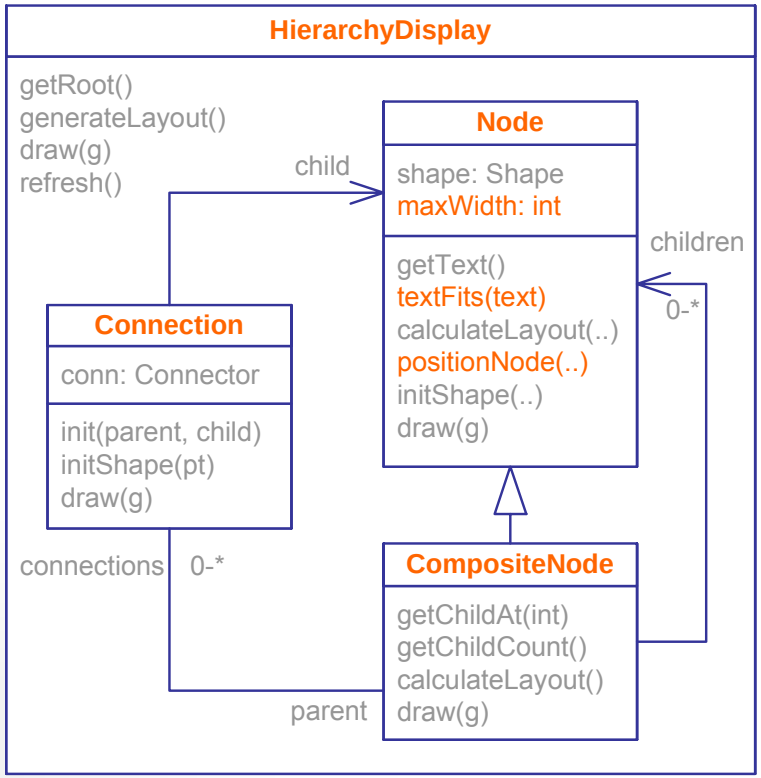
Extending with Virtual Classes



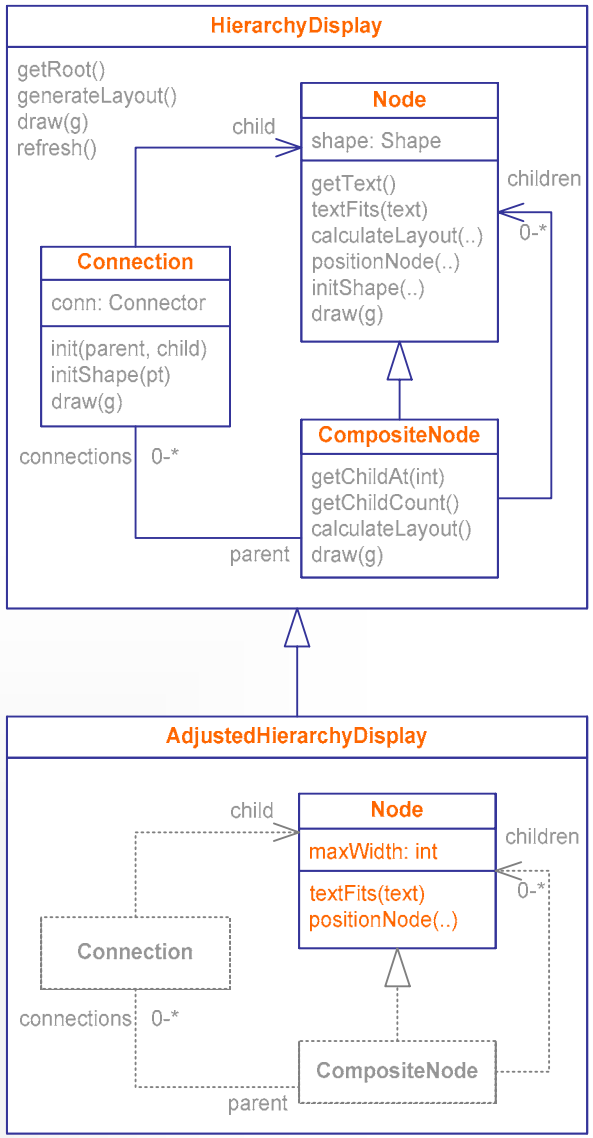
Virtual Classes



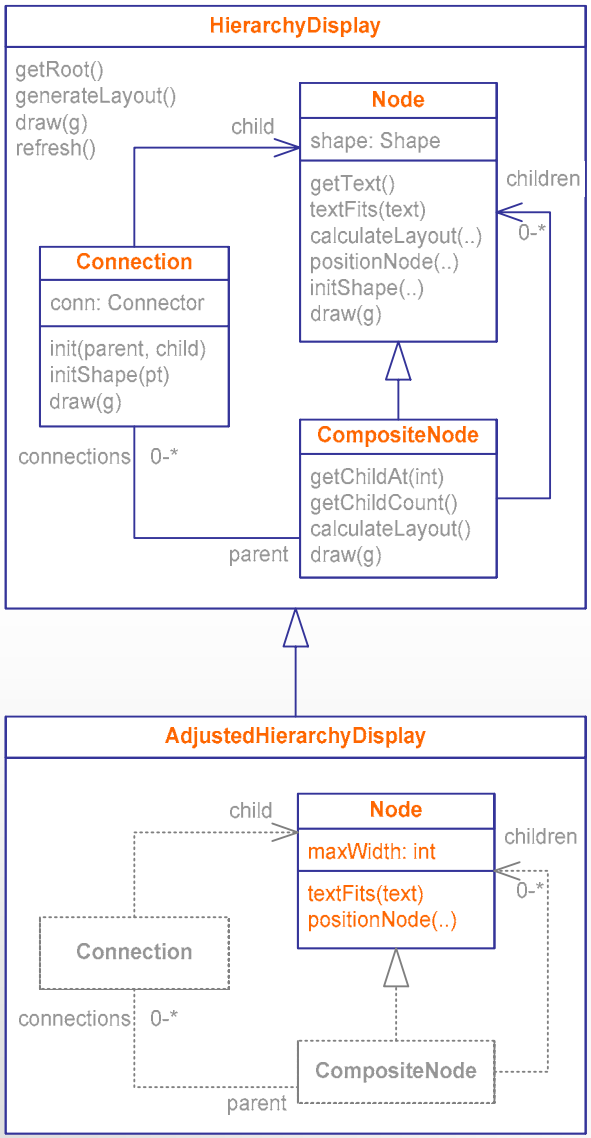
Virtual Classes



Virtual Classes



Virtual Classes



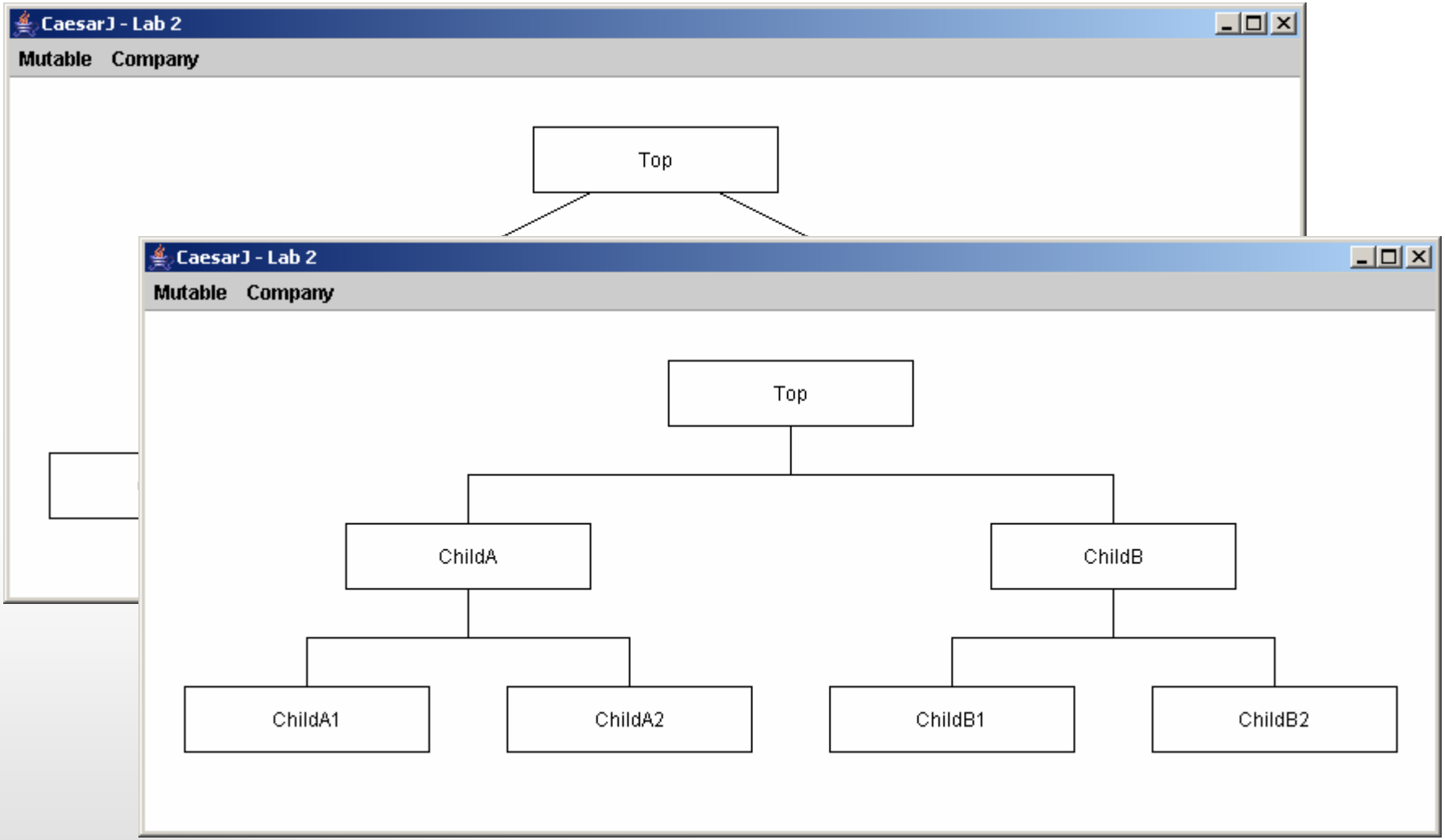
- Modules encapsulate several collaborating abstractions
- Abstractions are implemented in virtual classes
  - Can be overridden and late bound (just like virtual methods)
  - Old relations are inherited but automatically re-directed to the most specific definition of a type reference
  - New classes and relations can be added

Dependent Types

- Class families can be instantiated and used polymorphically
- Type safety is ensured by “path-dependent types”
  - Instances of virtual classes are compatible only if they belong to the same family
- The type system is formalized and its soundness is proved

```
final HierarchyDisplay hd = new AdjustedHierarchyDisplay();
final HierarchyDisplay hd2 = new AngularHierarchyDisplay();
hd.Node n = hd.new Node(); // ok
hd.foo(n); // ok
hd2.Node n = hd.new Node(); // static error
hd2.foo(n); // static error
```

Extension B: Change Connector Type



To Do: Change Connector Type

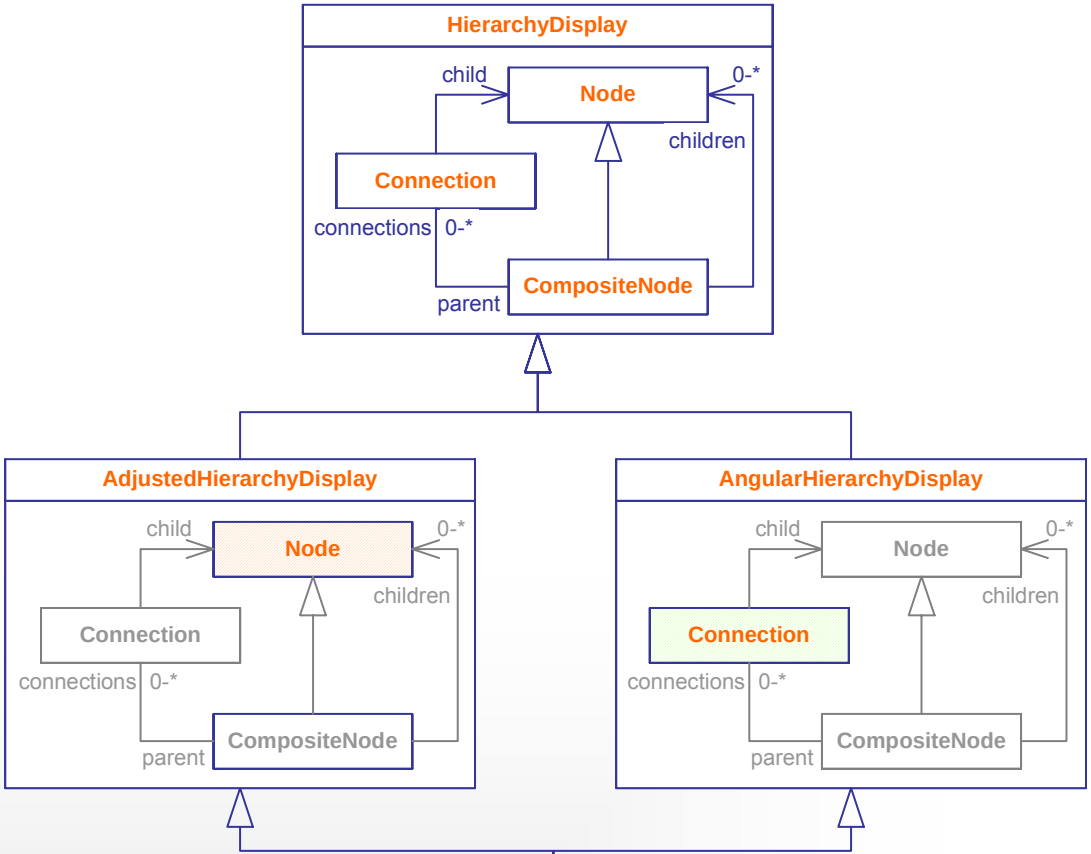
Task:

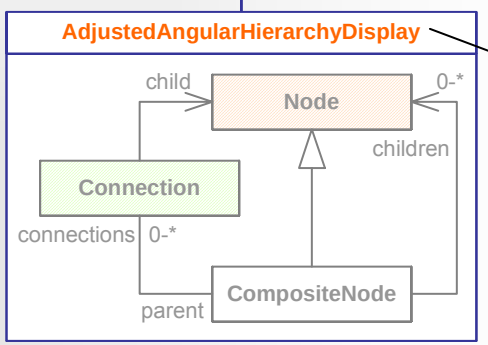
Extend hierarchy display component to use angled connectors.

Steps:

1. Open **`LabC`**
2. Declare **`AngularHierarchyDisplay`** as extension of **`HierarchyDisplay`**
3. Override **`Connection`** class
4. Override **`initShape()`** method
5. Implement it using **`RightAngledConnector`**
6. Implement **`showAngularHierarchy()`** method in **`HierarchyDisplayControl`**.

Composition of Extensions





- Modules can be combined using mixin composition semantics
- Composition propagates into virtual classes

```
cclass AdjustedAngularHierarchyDisplay
   extends AdjustedHierarchyDisplay &
           AngularHierarchyDisplay { }
```

To Do: Combine Extensions

Task:

Create a hierarchy display component with both extensions: nodes adjusted to text size and angled connectors.

Steps:

1. Declare **`AdjustedAngularHierarchyDisplay`** as combination of **`AdjustedHierarchyDisplay`** and **`AngularHierarchyDisplay`**

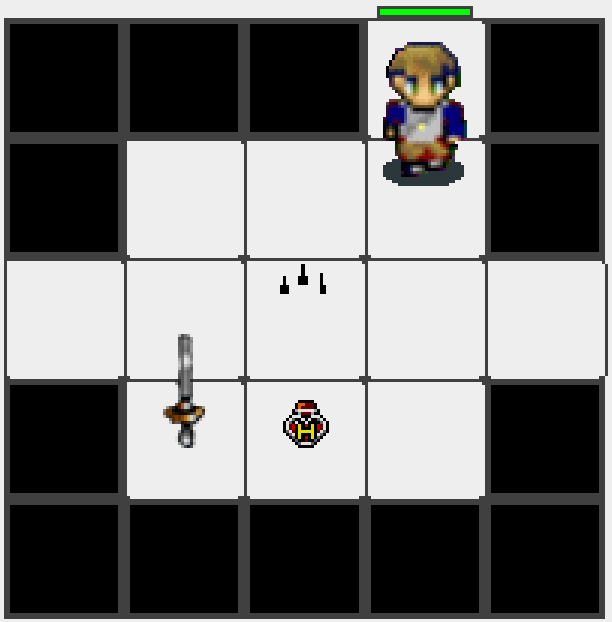
2. Implement **`showAdjustedAngularHierarchy()`** method in **`HierarchyDisplayControl`**.

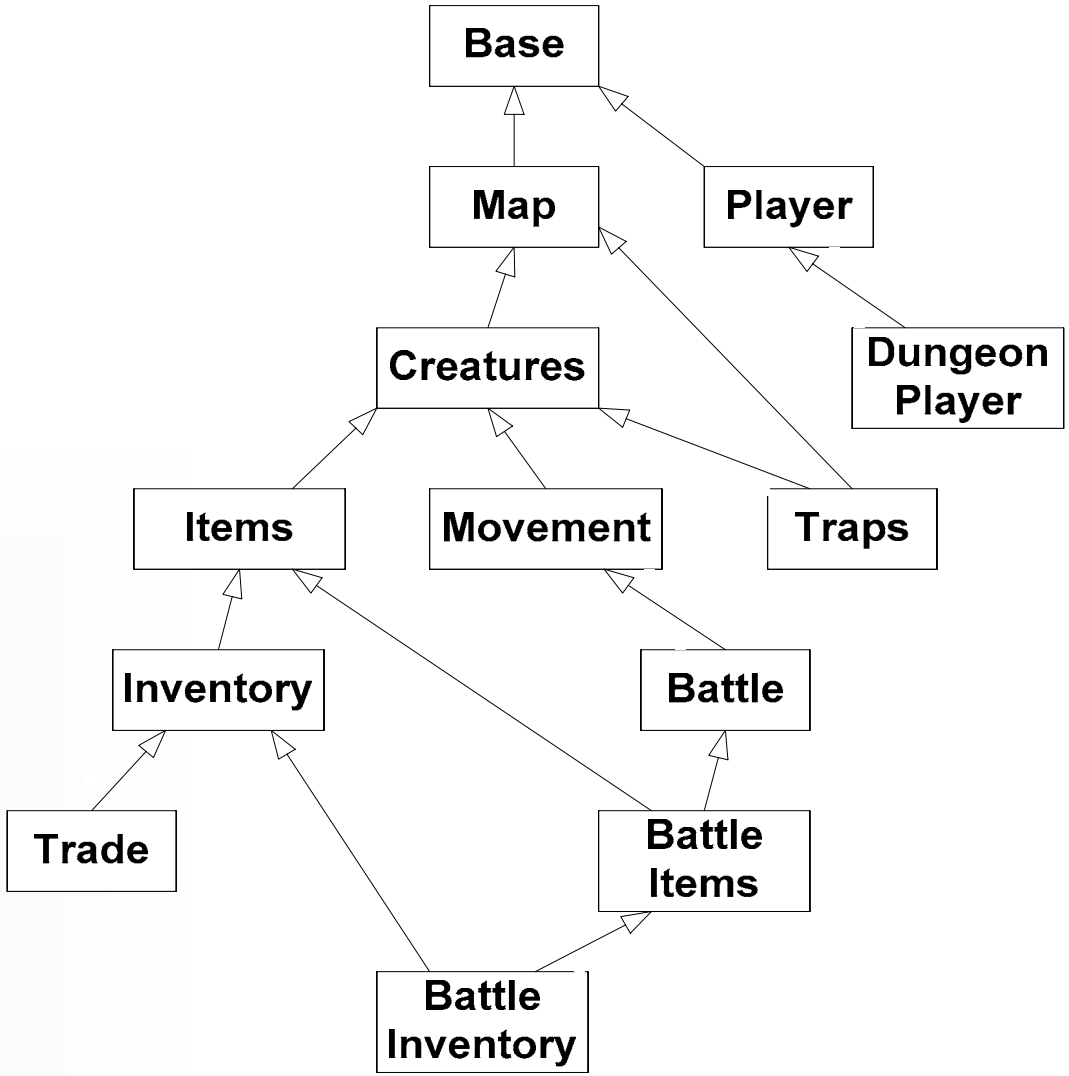
Feature-Based Modularization

- Typical extensions are new features
- A new feature does not only add new classes, but also changes existing ones
  - Thus a feature can be modularized as a layer that refines existing classes as well as add new ones
- In fact, the entire functionality of a component can be decomposed into such feature modules
- Concrete components can be created as mixin compositions of various sets of features

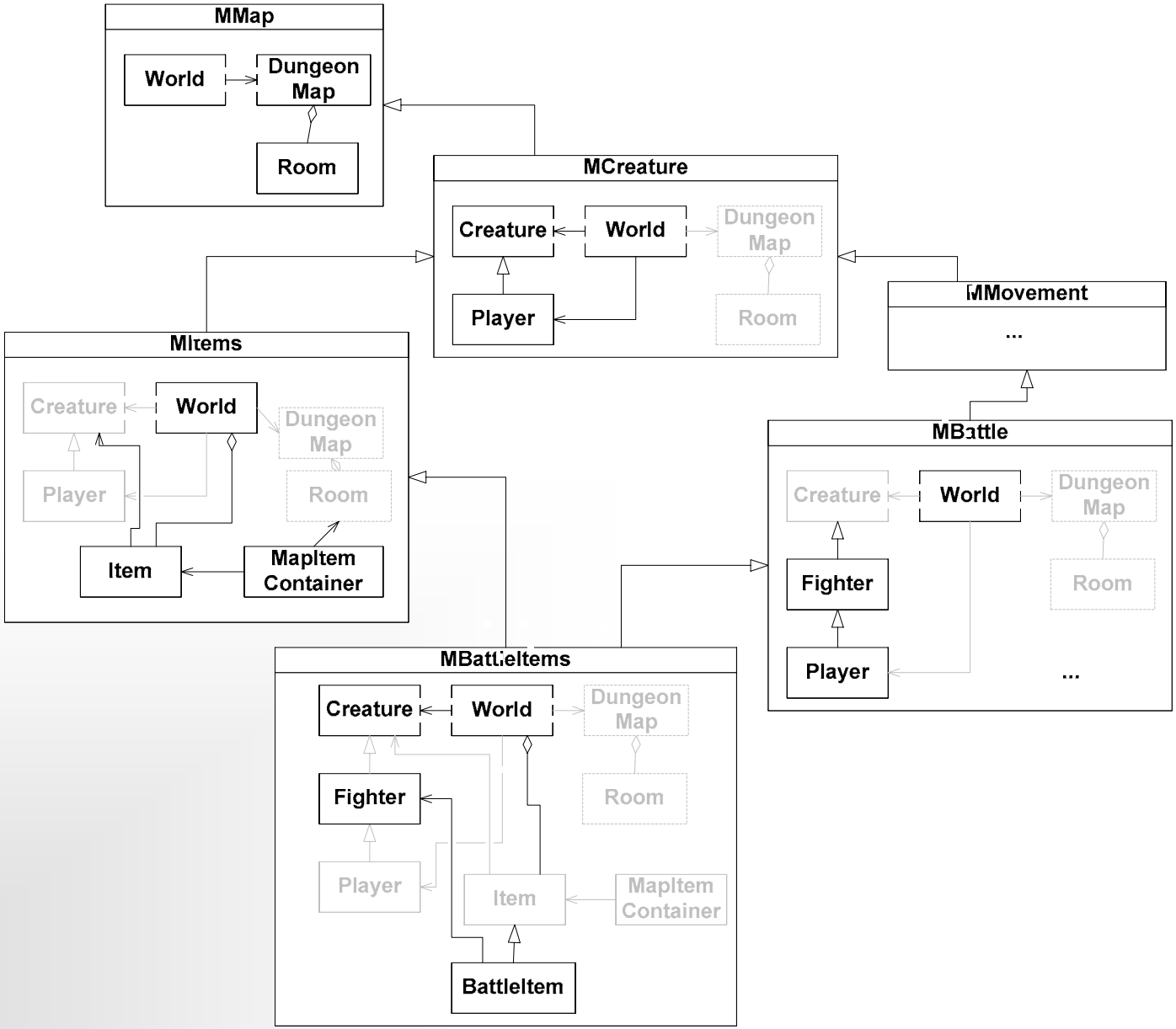
Feature-Based Modularization

- We can decompose the entire application into features





Feature-Based Modularization



Feature-Based Modularization

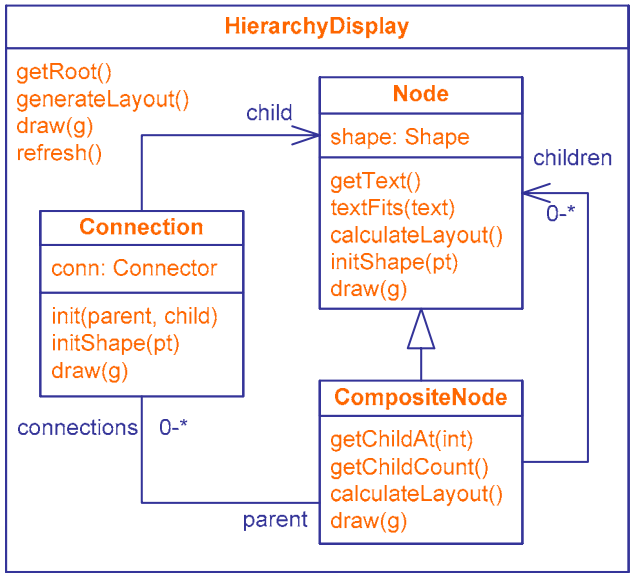
```
public cclass DungeonApp extends
    MGameOverRenderer
  & MItemInventoryUseControl
  & MPlayerMovementControl
  & MBattleControl
  & MGameEndBattleWorld
  & MGameEndFindPrincessWorld
  & MBattleRegeneration
  & MBattleItemInventoryRenderer
  & MItemInventoryRenderer
  & MBattleRenderer
  & MCreatureRenderer
  & MTrapsRenderer
  & MTraps
  & MItemRenderer
  & MMapRenderer
  & MCentricRoomCoords
  & MGetCurrentRoomByPlayer
  & MWorldFactoryItems
  & MWorldFactoryBattle
  & ...
```

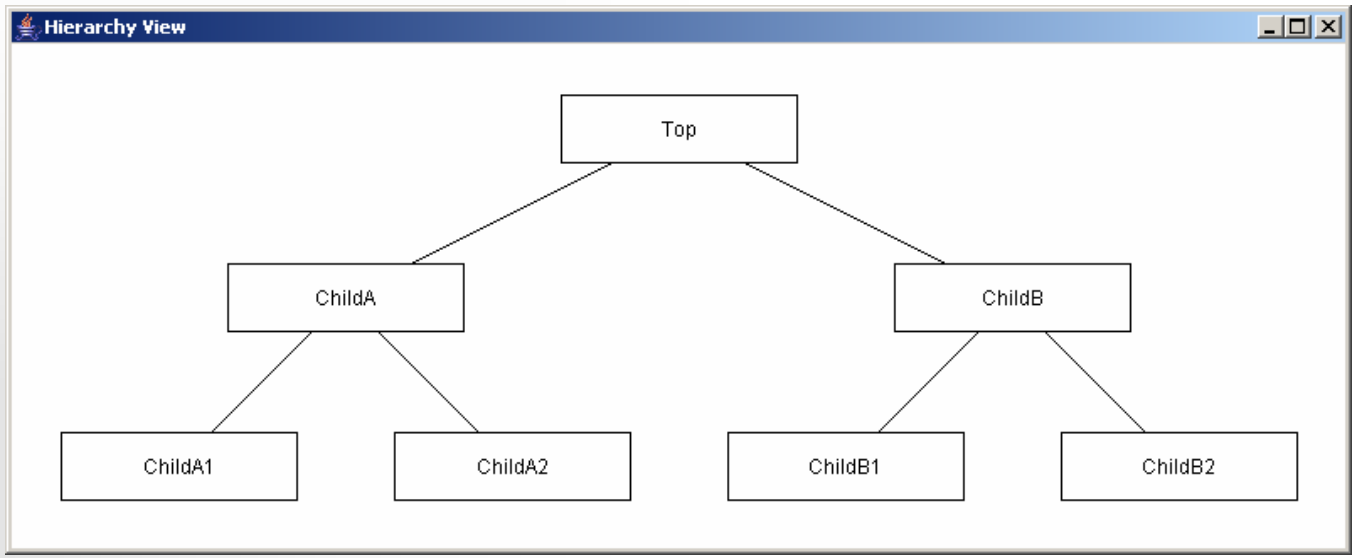
Features can be switched off or replaced by alternative features by changing the composition

Outline

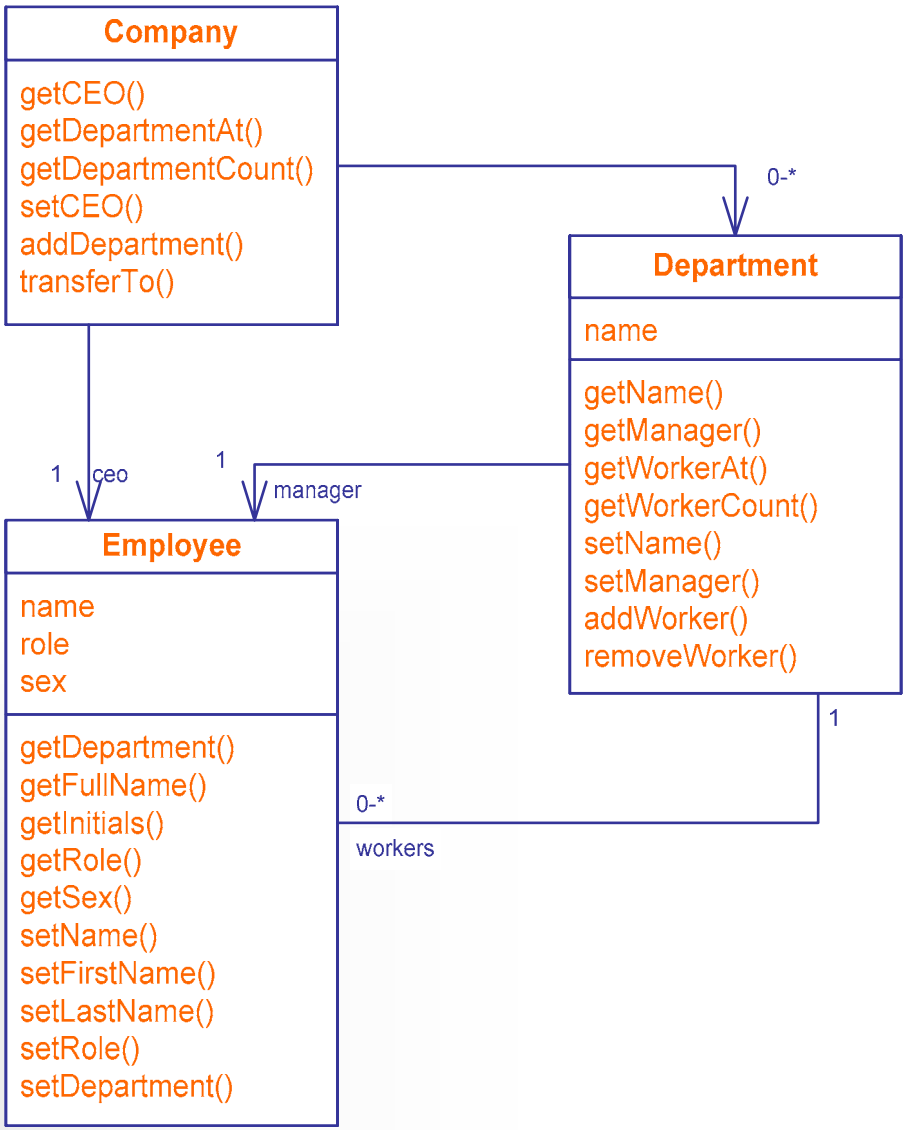
- Aspects in CaesarJ
  - Aspects as classes
  - Dynamic aspect deployment
- Hierarchical Refinements
  - Extending component with virtual classes
  - Combining different extensions
  - Feature-oriented decomposition
- **Crosscutting Integration**
  - **Defining wrapper classes**
  - **Observing events with pointcuts**
  - **Variability management**
- Integrating Distributed Components

Component A: Hierarchy Display

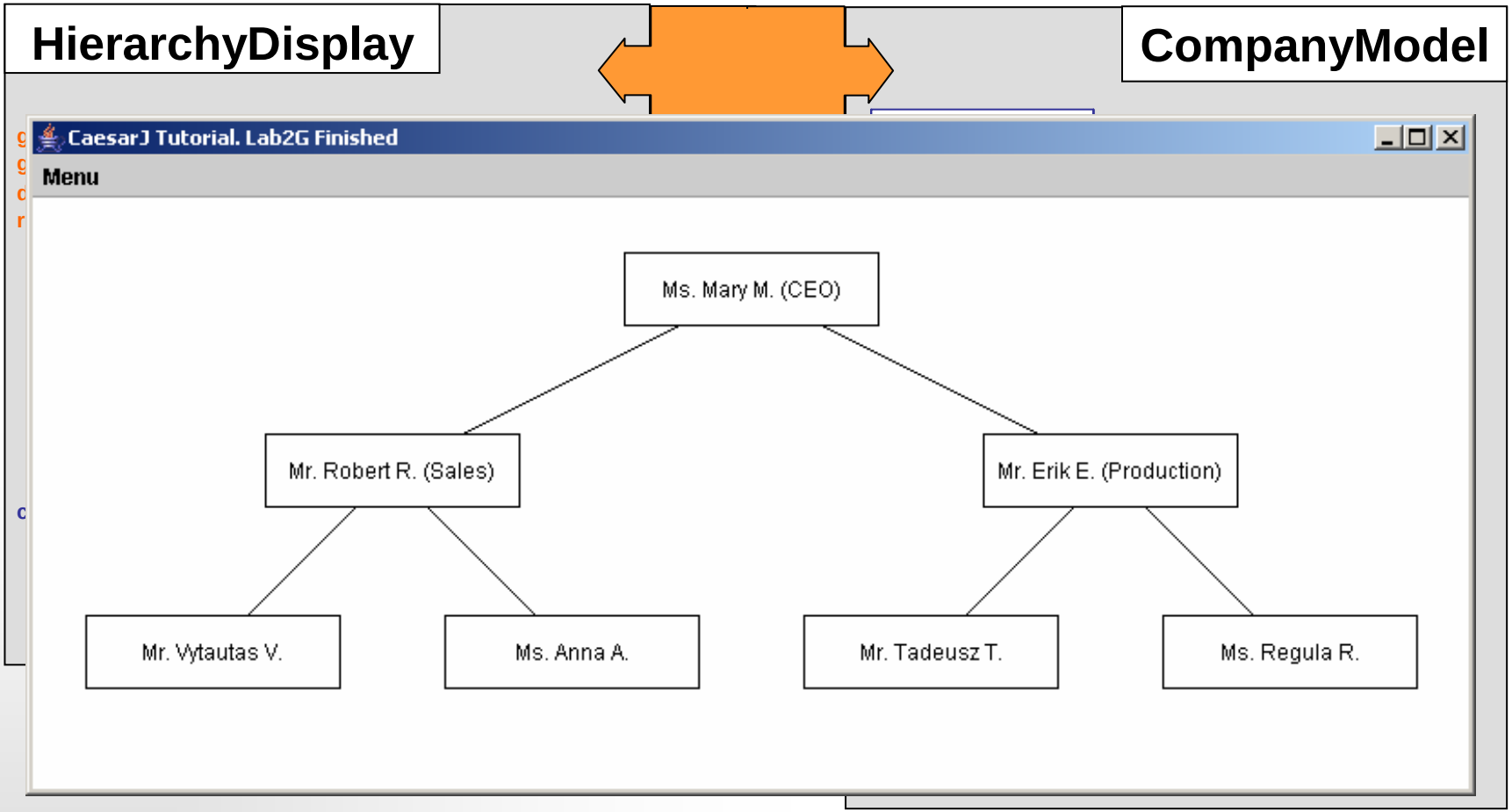




Application Data Model



Component Integration

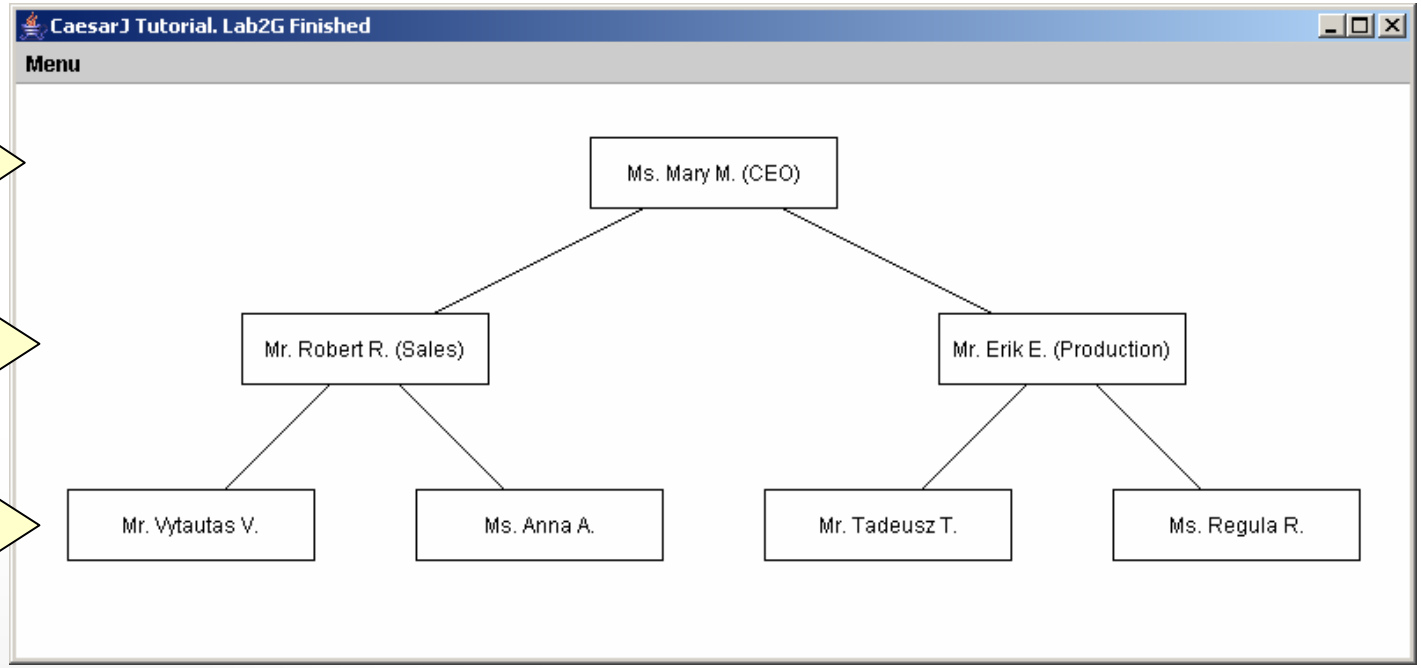


Component Integration

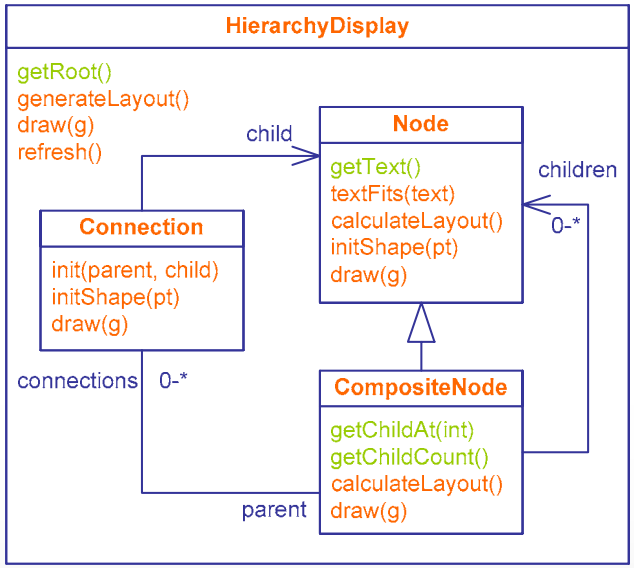
CEO

Department Managers

Workers

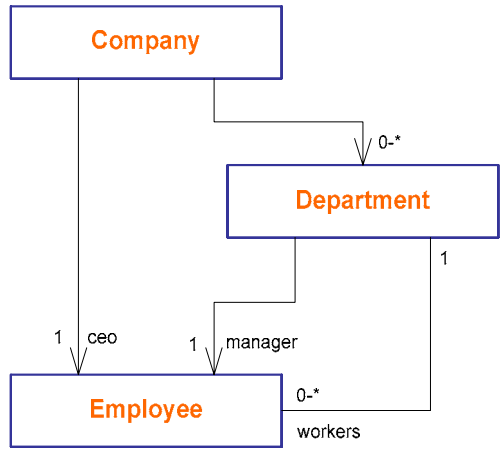


Integrating Components



Display uses data and relationships of company objects

structural mapping

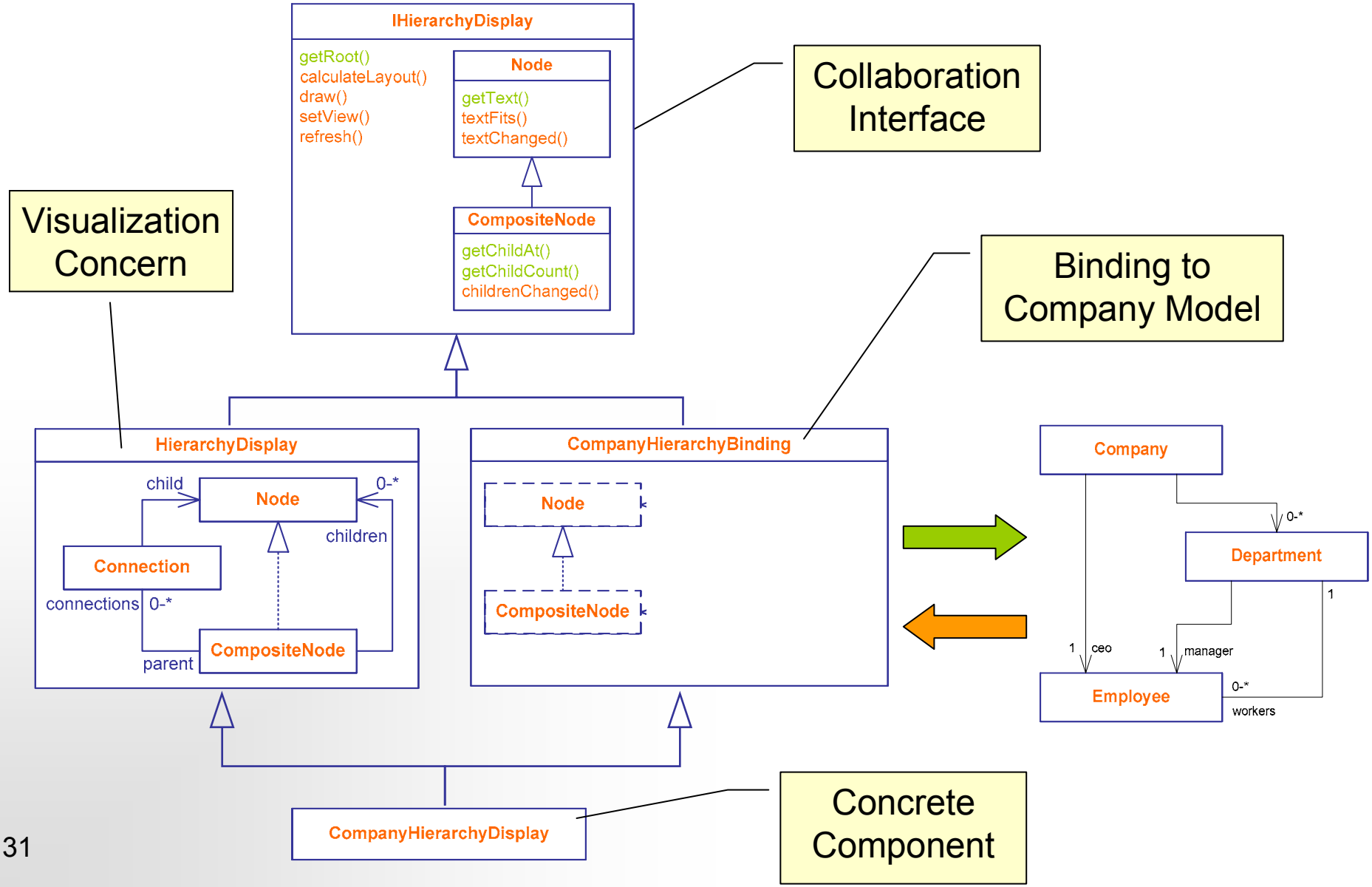


behavioral mapping

After certain changes display must be refreshed or layout recalculated

Design Goal:
Keep components independent of each other

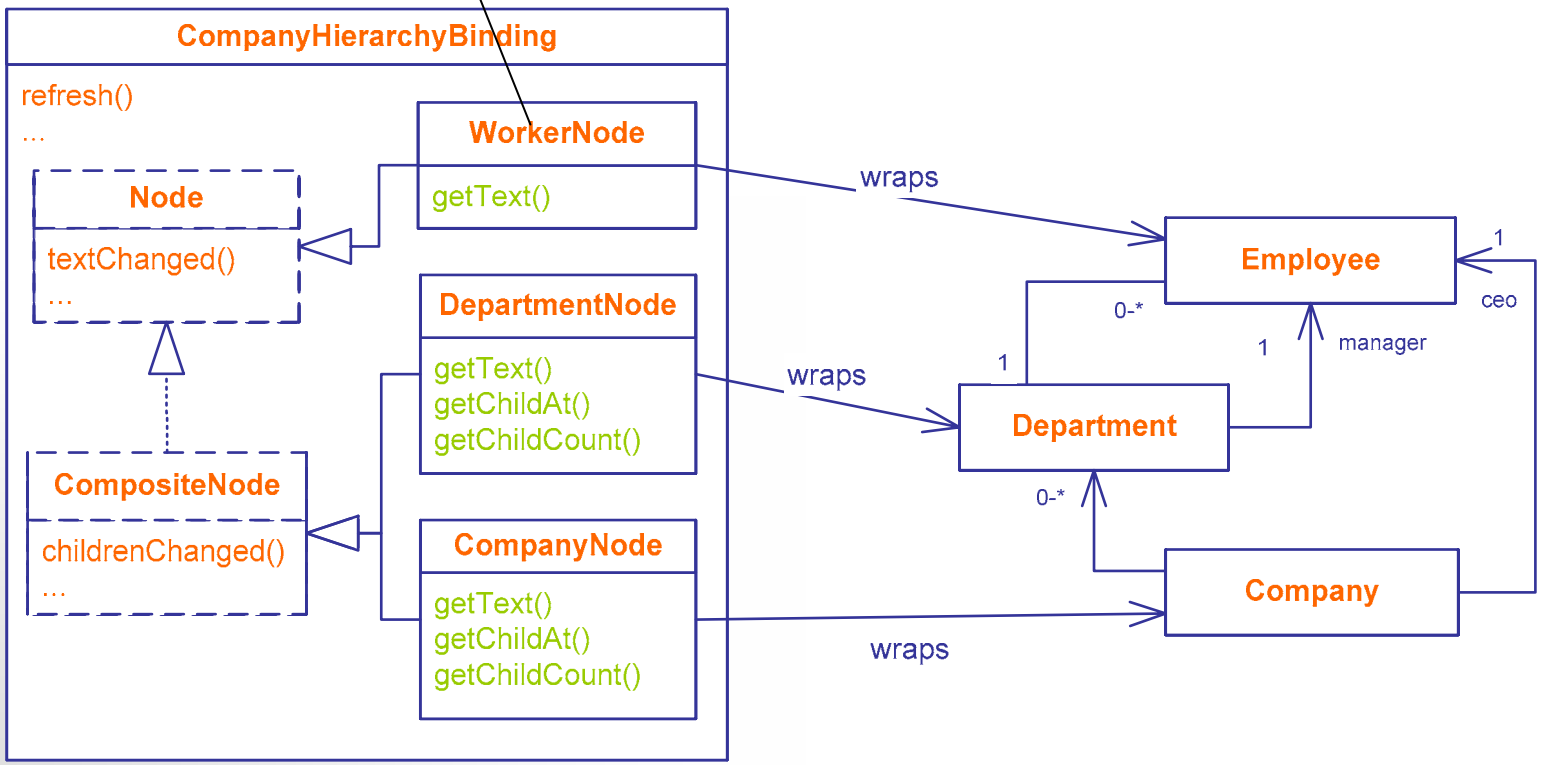
Component Integration



Component Integration - Bindings

```
cclass WorkerNode extends Node wraps Employee
{
    String getText() {
        return wrappee.getFullName();
    }
}
```

Adapting company model objects to visual nodes



Wrapper Recycling

```
cclass CompanyHierarchyBinding {
  cclass WorkerNode
    extends Node wraps Employee { ... }
  ...
}
```

```
void test() {
  CompanyHierarchyBinding hier =
    new CompanyHierarchyDisplay();

  Employee anna = new Employee();

  assert(hier.WorkerNode(anna) ==
         hier.WorkerNode(anna));

  Employee peter = new Employee();

  assert(hier.WorkerNode(anna) !=
         hier.WorkerNode(peter));
}
```

wrapper is created on demand

wrapper is reused for the same object

wrapper is destroyed by garbage collector

To Do: Extend Company Hierarchy

Task:

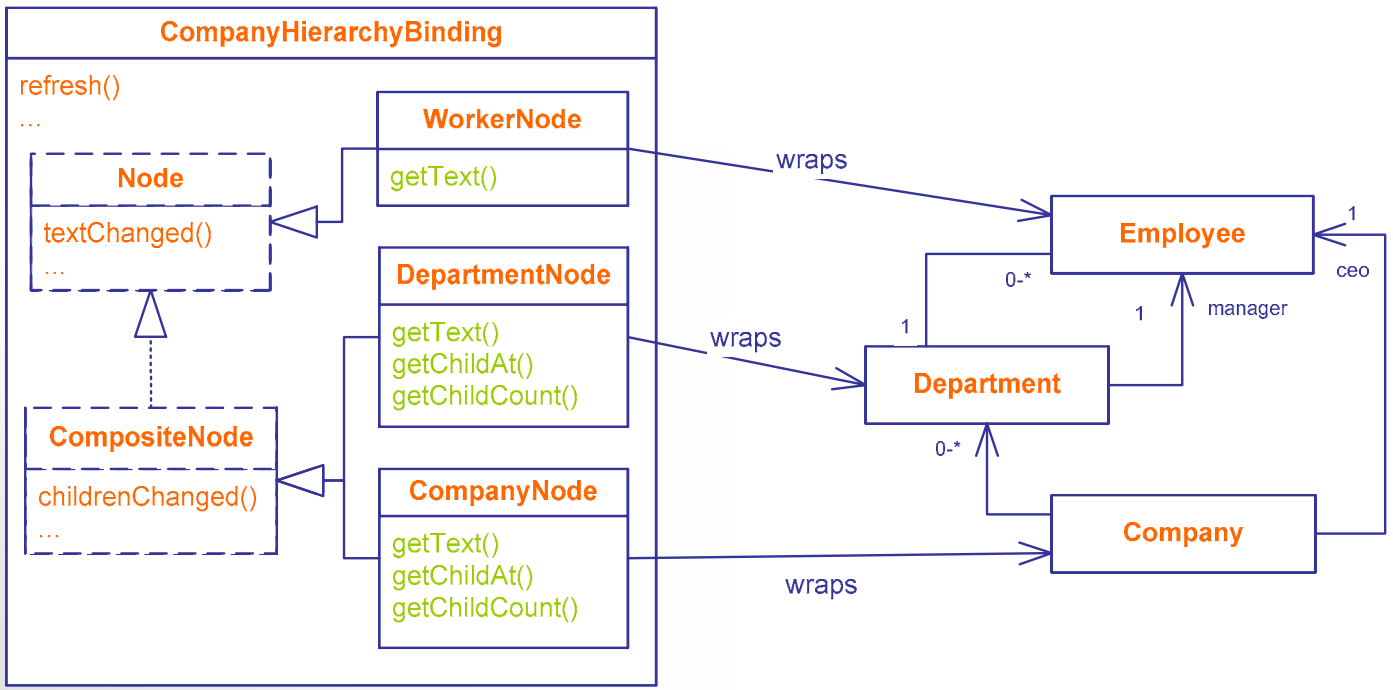
Extend company hierarchy binding with nodes to display department workers

Steps:

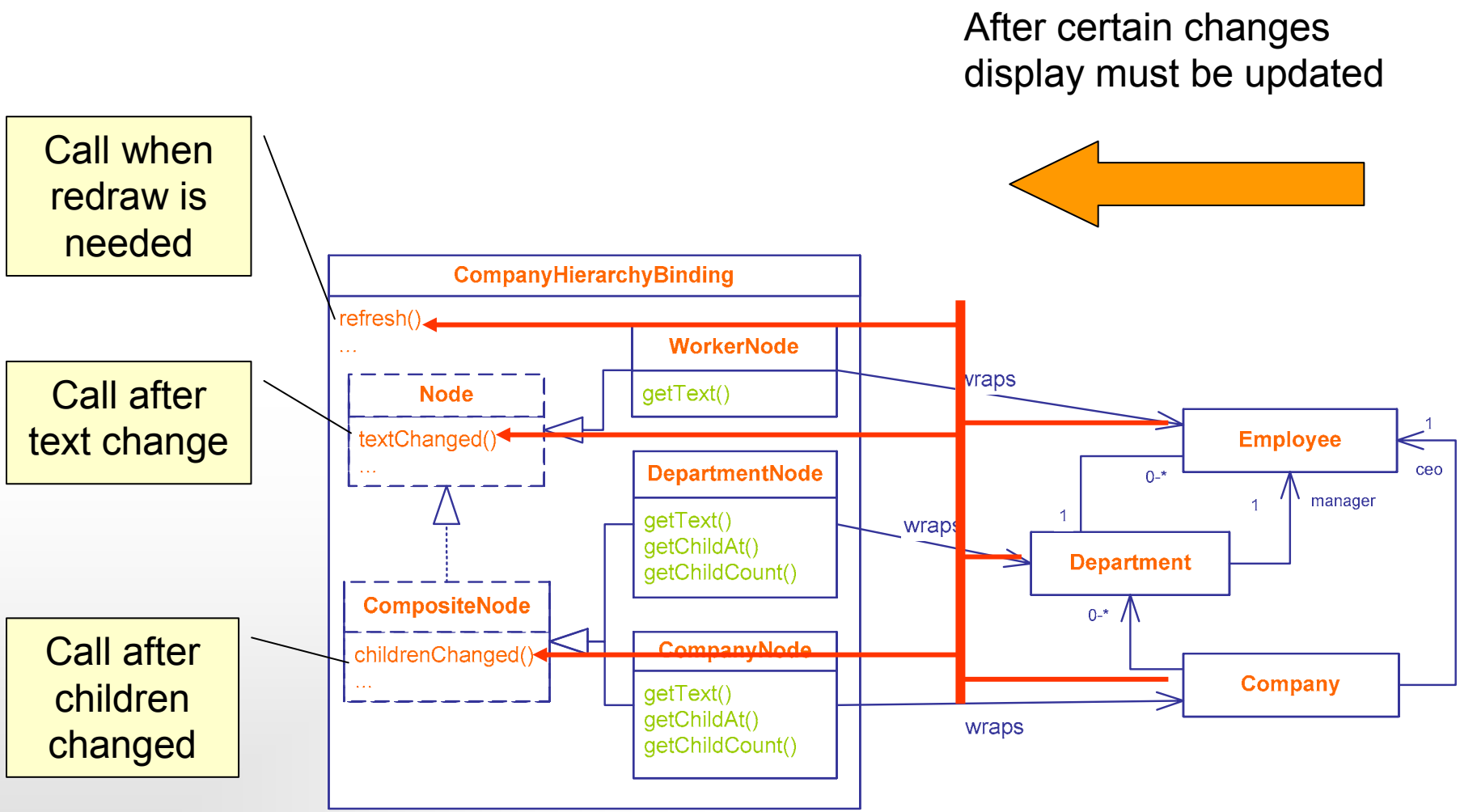
1. Open **`LabD`**
2. Declare **`WorkerNode`** wrapper class
3. Implement **`getText()`** using methods of **`wrappee`**
4. Declare **`DeparmentNode`** as subtype of **`CompositeNode`**
5. Implement **`getChildAt()`** and **`getChildCount()`** for **`DepartmentNode`**

Component Integration - Pointcuts

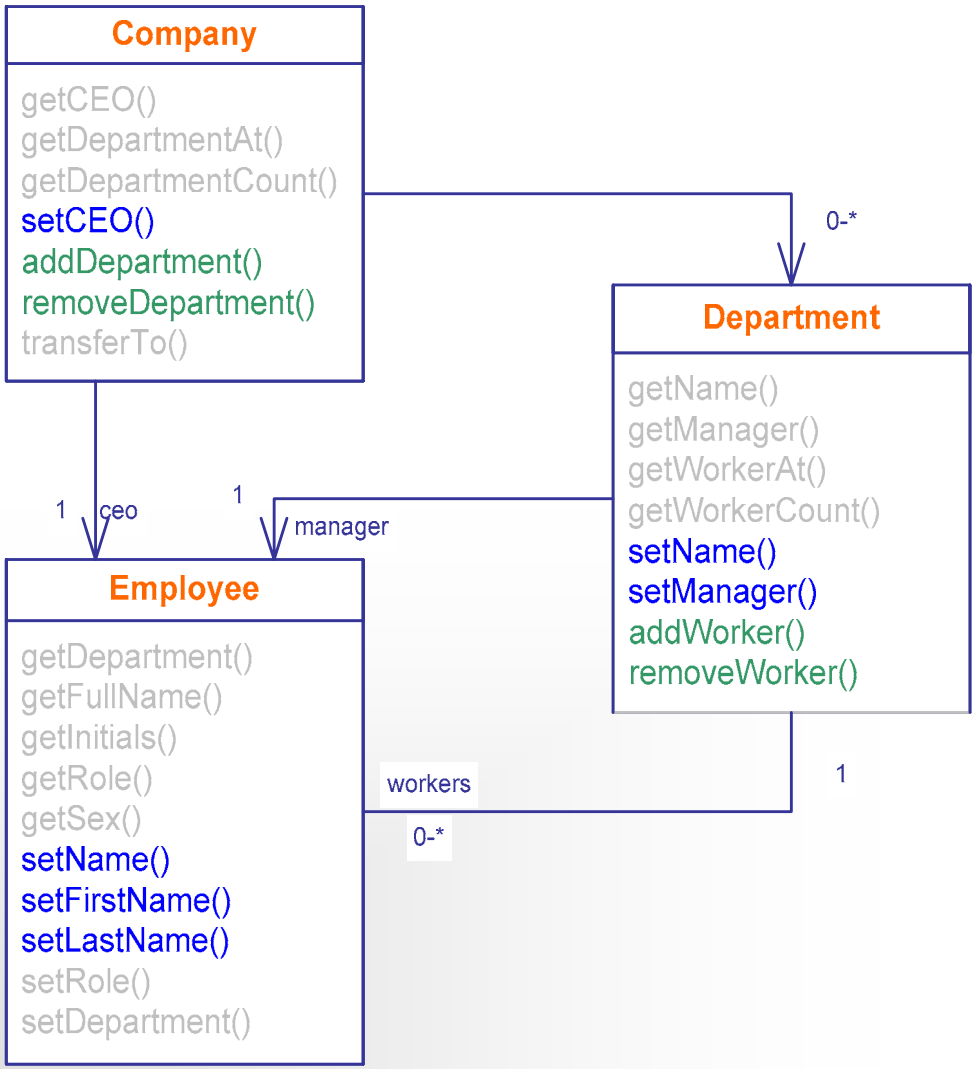
After certain changes
display must be updated



Component Integration - Pointcuts



Changes in Company Model



Affects text of a node

Affects children of a node

No direct effect

To Do: Observation with Pointcuts

Task:

Complete pointcuts and advice to observe text changes of worker nodes and children changes of department nodes.

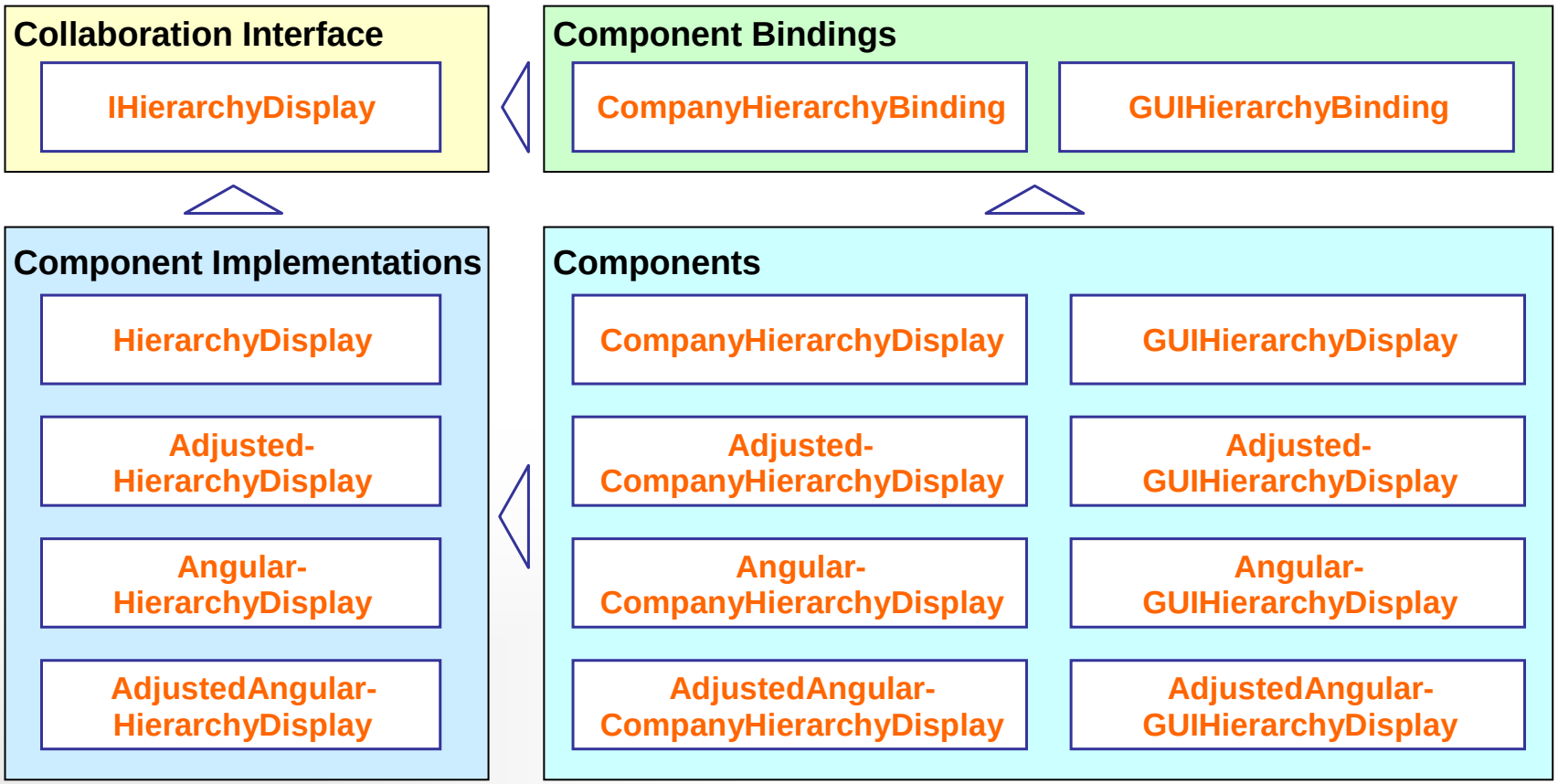
Steps:

- Open **`LabE`**
- In **`CompanyHierarchyBinding`** extend advice after Employee name change to update corresponding **`WorkerNode`**
- Write new advice with pointcut to observe changes of **`DepartmentNode`** children

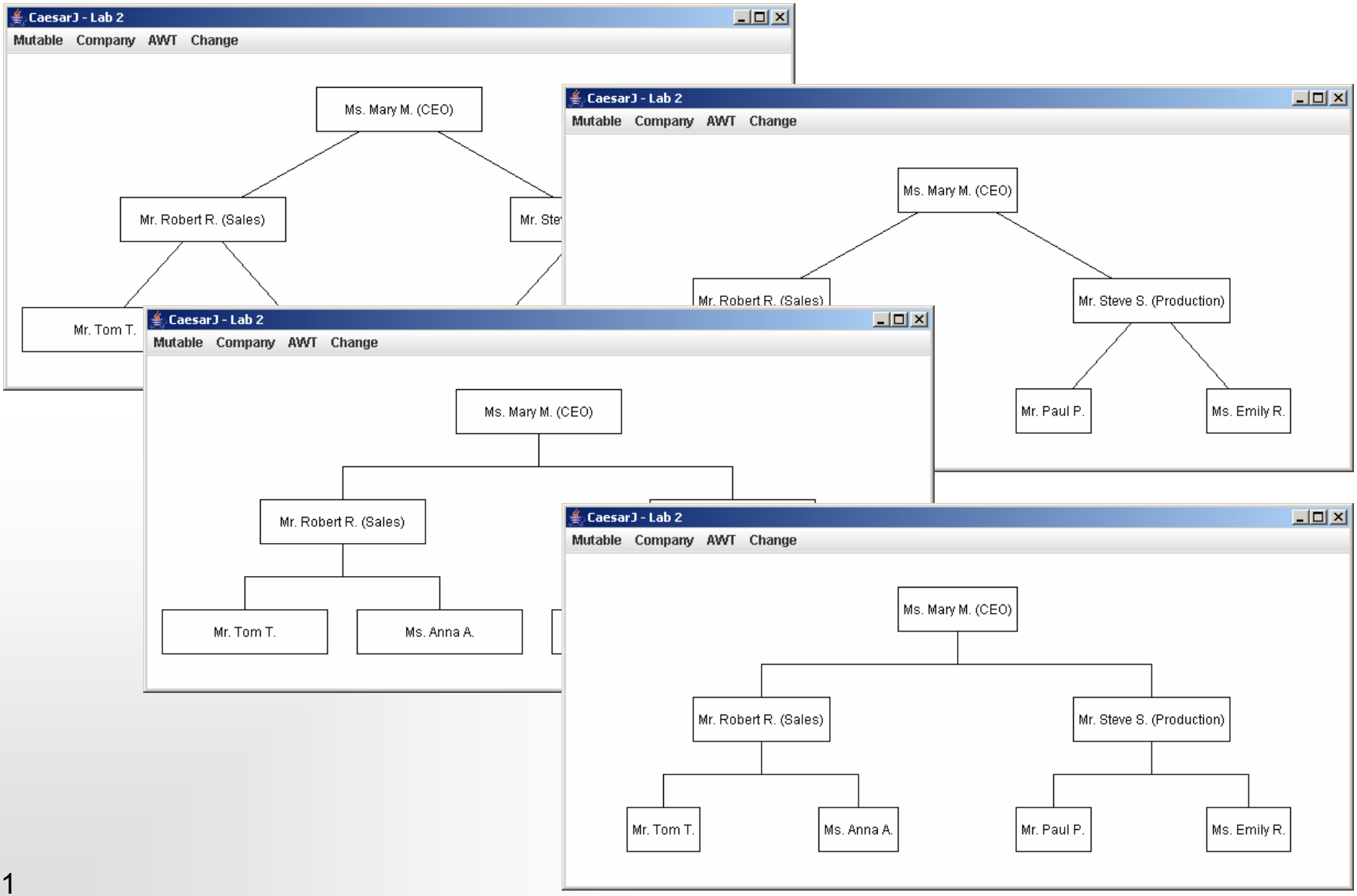
Component Integration

- Why didn't we use virtual classes for the integration?
- The **composition** of the two components **is not hierarchical**
  - The integrated classes have different names and exist independently from each other
  - One component class can be integrated with multiple classes of another component
- **Behavioral integration** is crosscutting
  - The behavior of a component is triggering at multiple points of the execution of the other component
  - It is not a logical part of a component functionality to trigger the behavior of other components

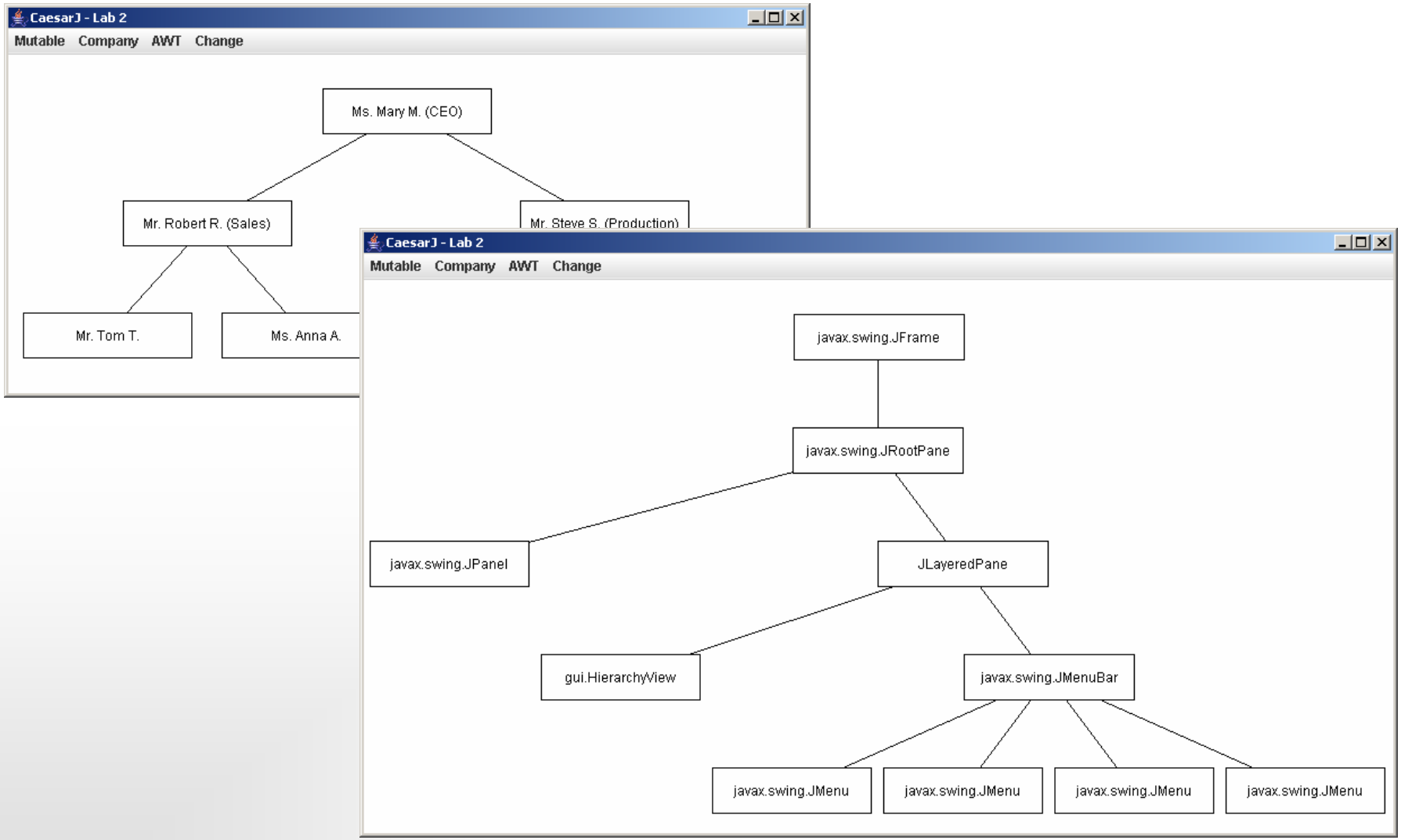
Combining Variations



Component Implementation Variations



Component Binding Variations



To Do: Vary Concrete Combinations

Task:

Combine company hierarchy and GUI hierarchy bindings with different hierarchy display implementations

Steps:

1. Open **`LabF`**
2. Define new classes as combinations of **`CompanyHierarchyBinding`** with various refinements of **`HierarchyDisplay`**
3. Implement **`showXXXCompanyHierarchy()`** methods in **`HierarchyDisplayControl`**
4. Combine **`GUIHierarchyBinding`** binding with different **`HierarchyDisplay`** implementations.
5. Implement **`showXXXGUIHierarchy()`** methods in **`HierarchyDisplayControl`**

Dynamic Wrapper Selection

```
abstract public cclass GUIHierarchyBinding
                       extends IHierarchyDisplay {
   public cclass ComponentNode extends Node wraps Component
   { ... }

   public cclass ContainerNode extends CompositeNode
                                wraps Container {
      public Node getChildAt(int i1) {
         return wrapComponent(wrappee.getComponent(i1));
      }
      ...
   }
   public Node wrapComponent(Component comp) {
      if (comp instanceof Container) {
         return ContainerNode((Container)comp);
      }
      else {
         return ComponentNode(comp);
      }
   }
}
```

Wrappers are selected depending on the dynamic type of the wrappee

Dynamic Wrapper Selection

```
abstract public cclass GUIHierarchyBinding
                       extends IHierarchyDisplay {
   public cclass ComponentNode extends Node wraps Component
   { ... }

   public cclass ComponentNode extends CompositeNode
                               wraps Container {
      public Node getChildAt(int i1) {
         return ComponentNode(wrappee.getComponent(i1));
      }
      ...
   }
}
```

- We can override **`ComponentNode`** for **`Container`**
- But what if we override **`ComponentNode`** in subclasses of **`GUIHierarchyBinding`**?
- What is meaning of the type **`this.ComponentNode`**?

Multidimensional Virtual Classes

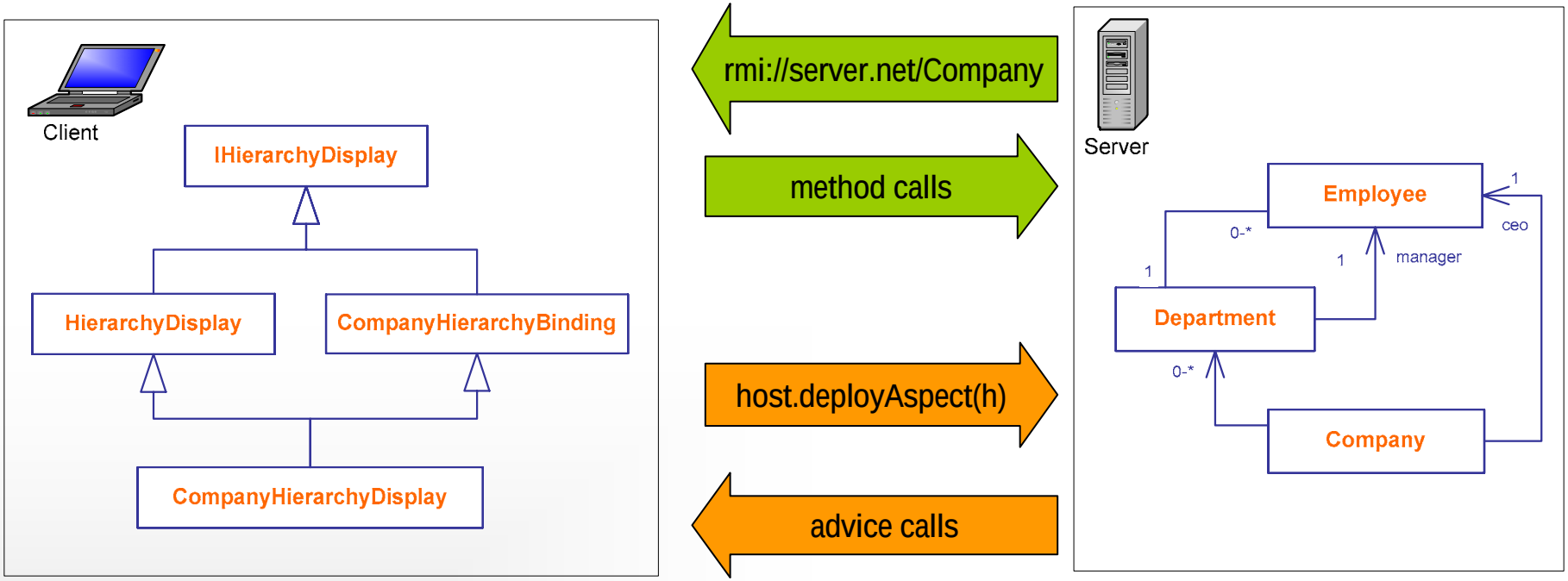
- Dynamic wrappers are a special case of a more general concept: **multidimensional virtual classes**
  - Virtual classes defined relative to multiple objects

```
cclass GUIHierarchyBinding extends IHierarchyDisplay

cclass ComponentNode <GUIHierarchyBinding h, Component w>
{ ... }

cclass ComponentNode<GUIHierarchyBinding h, Container w>
{
   public Node<h> getChildAt(int i1) {
      return new ComponentNode<h, wrappee.getComponent(i1)>;
   }
   ...
}
```

Outline

- Aspects in CaesarJ
  - Aspects as classes
  - Dynamic aspect deployment
- Hierarchical Refinements
  - Extending component with virtual classes
  - Combining different extensions
  - Feature-oriented decomposition
- Crosscutting Integration
  - Defining wrapper classes
  - Observing events with pointcuts
  - Variability management
- **Integrating Distributed Components**

Integrating Distributed Components



Remote Caesar Classes

Transparent usage:

```
cclass Company {
   public Employee getCEO() {
      return ceo;
   }
   ...
}

CaesarHost host = new CaesarHost(“rmi://localhost/”);
Company company = (Company)host.resolve("Company");
String ceoName = company.getCEO().getName();
...
```

Caesar RMI Compiler

```
<target name="rmic" depends="compile">
  <java classname="org.caesarj.rmi.Compiler">
    <classpath refid="caesar.classpath"/>
    <arg value="-d"/>
    <arg value="${destdir}"/>
    <arg value="-c"/>
    <arg value="${projrmiclasspath}"/>
    <arg value="-r"/>
    <arg value="company.Company"/>
    <arg value="company.Department"/>
    <arg value="company.Emploee"/>
    <arg value="hierarchies.company.CompanyHierarchyDisplay"/>
    <arg value="hierarchies.company.AdjustedCompanyHierarchyDisplay"/>
    <arg value="hierarchies.company.AngularCompanyHierarchyDisplay"/>
    <arg value="hierarchies.company.AdjustedAngularCompanyHierarchyDisplay"/>
  </java>
</target>
```

- Prepares Caesar classes
- Use option `-r` of to prepare inner classes
- Use standard RMI compiler for Java classes

Remote Aspect Deployment

Server side:

```
CaesarHost host = new CaesarHost(“rmi://localhost/”);
host.activateAspectDeployment();
```

Client side:

```
CaesarHost host = new CaesarHost(“rmi://localhost/”);
ICompanyHierarchyDisplay hier
                  = new CompanyHierarchyDisplay();
host.deployAspect(hier);
...
host.undeployAspect(hier);
```

To Do: Display Remote Company Model

Task:

Use different colors to display different types of company hierarchy nodes.

Steps:

- Open **`LabG`** project
- Change deployment strategy of hierarchy objects to remote deployment
- Enable aspect deployment on server side
- Build project using **`build.xml`** script
- Run **`ServerMain`**
- Run **`Application`**

Thank you!

http://caesarj.org