Overview of CaesarJ

Eduardo Figueiredo

http://www.dcc.ufmg.br/~figueiredo

What is CaesarJ?

- A programming language for
  - Aspect-Oriented Programming
  - Feature-Oriented Programming
- Aims to improve software design
  - modularity, reusability, flexibility, etc.
- Fully integrated with Java
  - Compile to JVM-compatible bytecode

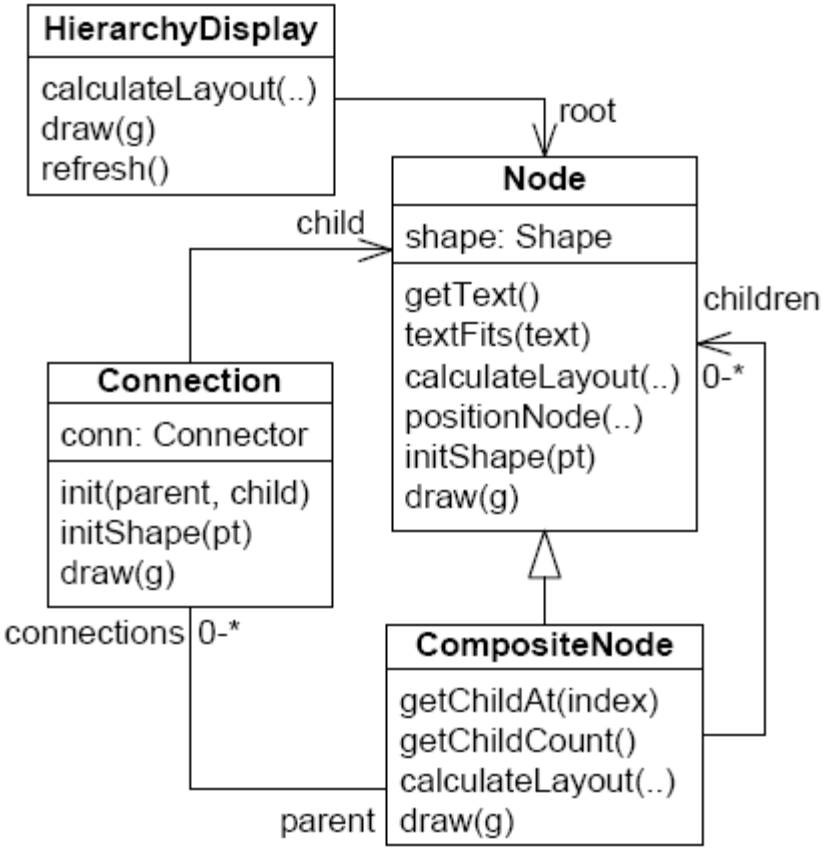
CaesarJ Highlights

- Family Class
- Virtual Class
- Mixin Composition
- Binding
- Collaboration Interface
- Dynamic Deployment
- Aspectual Polymorphism

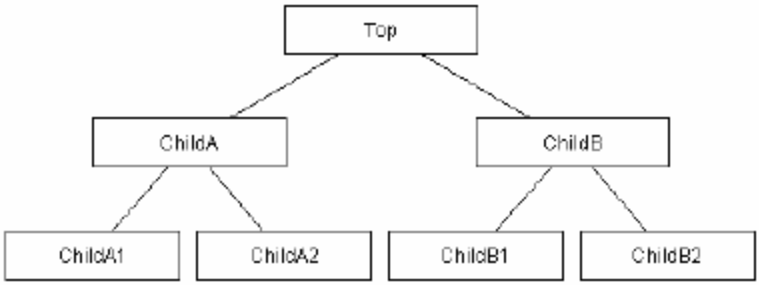
Family Class

- Also known as CaesarJ class
  - Or *cclass*, for short
- It groups related functionality
  - Or just *classes*
- Class is a recursive definition

Four Classes (OO)

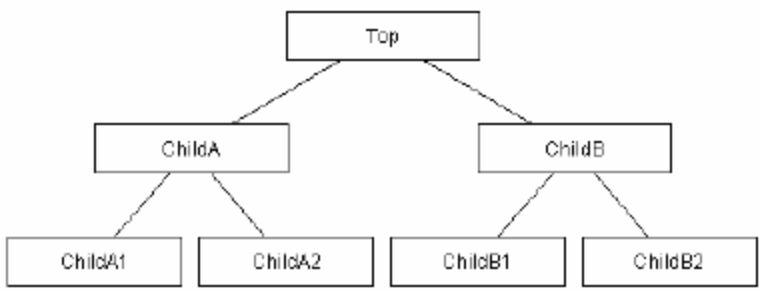


They represent a tree

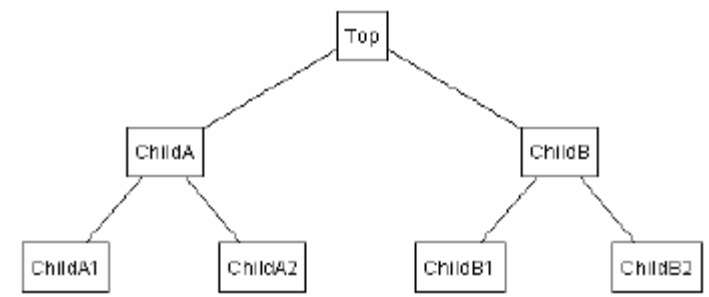


Variability Management (SPL)

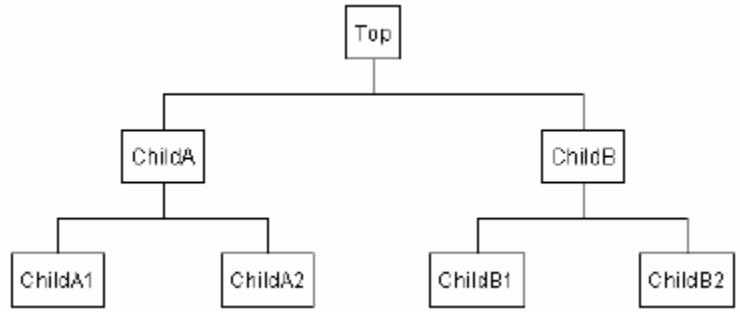
A basic tree



…with adjusted nodes



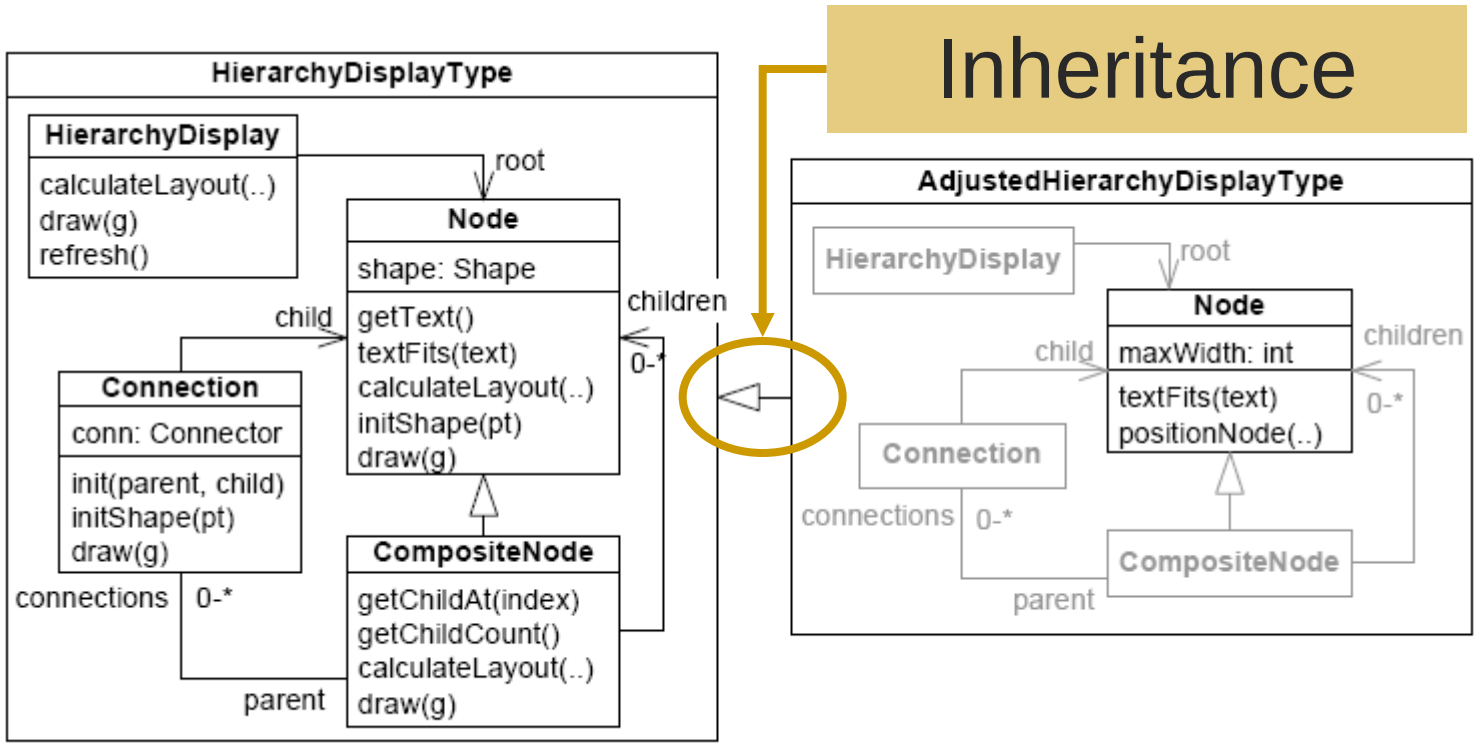
…plus angular connections



Virtual Classes

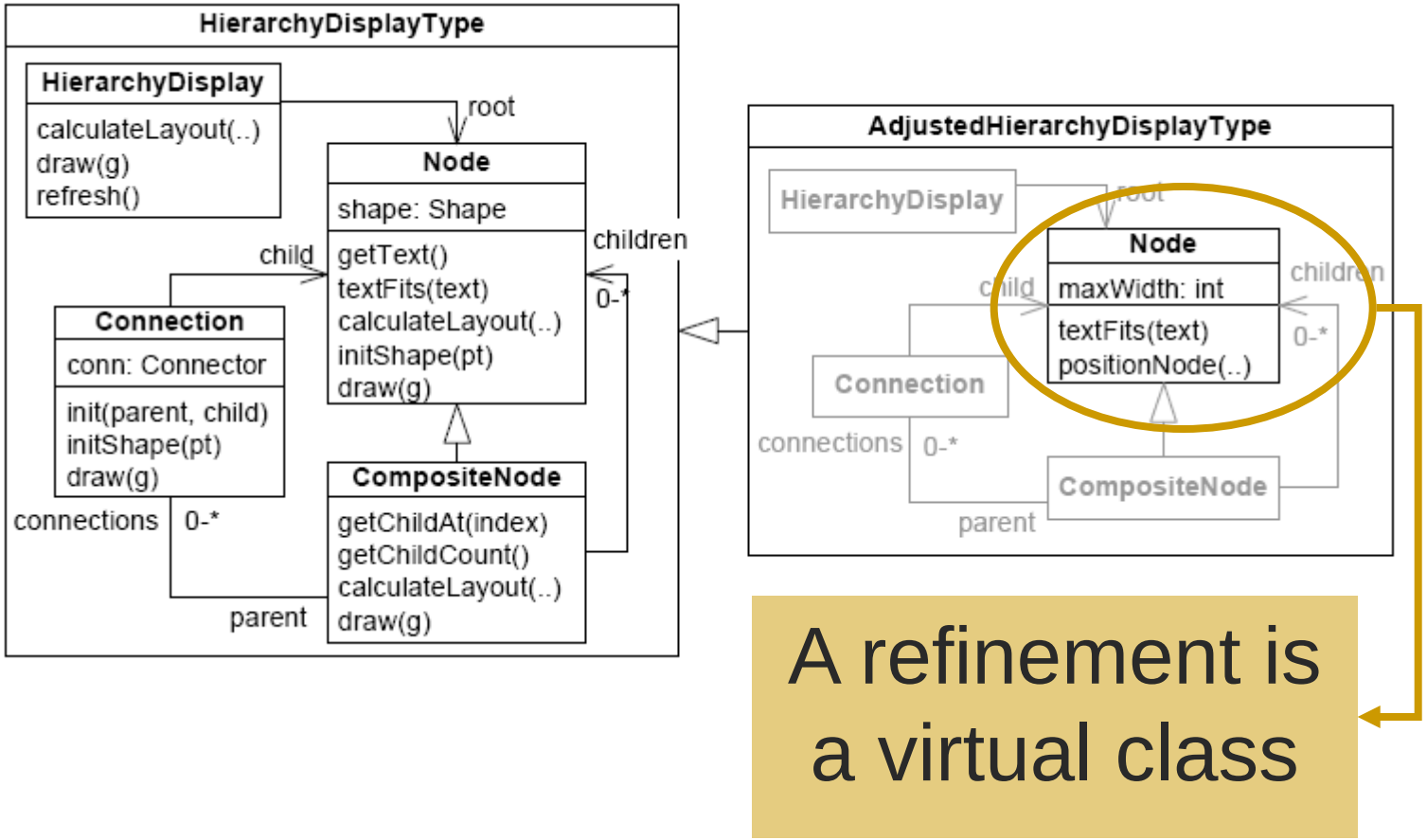
- They have different meanings depending on the dynamic context
  - Similar to virtual methods
- They are defined as inner classes of an enclosing family class (*cclass*)
  - Members of the family

Two Family Classes



A family class is also a class

Two Family Classes

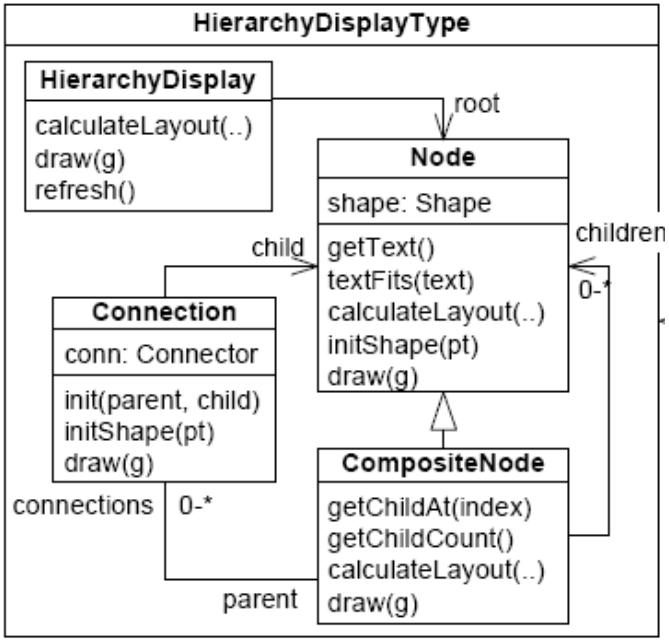


Mixin Composition

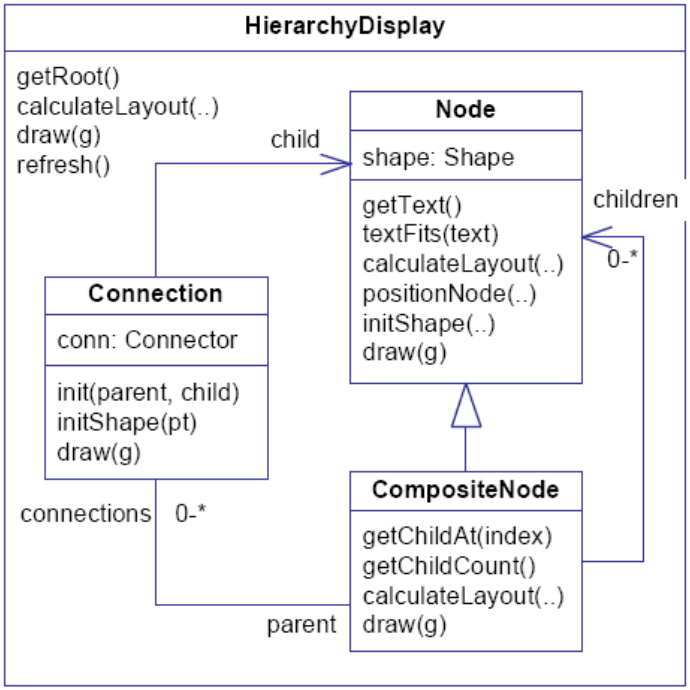
- Classes (simple or families) are mixins
  - Their superclass can be exchanged
- Mixin composition propagates into inner classes of a family class automatically
  - The display layout strategy can compose adjustable nodes with angular connections
- Unambiguousness is ensured by
  - the composition order
  - a linearization algorithm

Members of a Family Class

- In addition to virtual classes, a family class can also have members
  - Methods, attributes, etc.



=



HierarchyDisplay (CaesarJ)

```
cclass HierarchyDisplay {
  cclass Node { ... }
  cclass CompositeNode extends Node {
    calculateLayout() {
      Connection c = new Connection(); ...
    }
  ...
  }
  cclass Connection { ...
    void initShape(Point pt) { ... }
  }
  Node root;
  ...
}
```

Virtual Classes

Family Class

HierarchyDisplay (CaesarJ)

```
cclass HierarchyDisplay {
  cclass Node { ... }
  cclass CompositeNode extends Node {
    calculateLayout() {
      Connection c = new Connection(); ...
    }
  ...
  }
  cclass Connection { ...
    void initShape(Point pt) { ... }
  }
  Node root;
  ...
}
```

refers to

Scope rules are similar to Java

HierarchyDisplay Subclasses

```
cclass AdjustedHierarchyDisplay extends HierarchyDisplay {
  cclass Node { ...
    int maxwidth;
  }

  void foo(Node n) { ...
    n.maxwidth ...
  }
}

cclass AngularHierarchyDisplay extends HierarchyDisplay {
  cclass Connection {
    void initShape(Point pt) { ... }
    ...
  }
}
```

Maxwidth is a new attribute of Node

No casting is required because **n** refers to AdjustedHierarchyDisplay.Node

Binding

- CaesarJ provides two mechanisms for expressing crosscutting compositions
  - AspectJ-like pointcut/advice
  - Binding propagating mixin composition

- Example of binding

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

The & Operator

- A variant of multiple inheritance
  - Linearizes the superclasses avoiding ambiguities
- The operator & is not commutative
  - The left hand side is more specific
- It works recursively with arbitrary levels of nesting

Collaboration Interface

- The interface concept is not necessary
  - CaesarJ does not have the single inheritance bottleneck
  - An abstract class cannot be instantiated
- Collaboration interface is an abstract class that controls the collaboration between different facets of its implementation

Example: Collaboration Interface

```
abstract public cclass IHierarchyDisplay {
  abstract public Node getRoot();                    /* data model */
  abstract public void calculateLayout();
  abstract public void draw(Graphics g);             /* visualization */
  abstract public void refresh();
  ...
  abstract public cclass Node {
    abstract public String getText();                /* data model */
    abstract public boolean textFits(String text);   /* visualization */
  }
  abstract public cclass CompositeNode extends Node {
    abstract public Node getChildAt(int i);          /* data model */
    abstract public int getChildCount();
    abstract public void calculateLayout();          /* visualization */
  }
}
```

Dynamic Deployment

- Denoted by the keyword *deploy* used as a block statement
- Dynamic deployment is useful when we cannot determine which variant of a feature should be used until runtime
  - In static deployment (like in AspectJ), we have to implement all different variants by conditional logic within the aspect code

Example: Dynamic Deployment

```
cclass Logging {
  after(): (call(void Point.setX(int)) || call(void Point.setY(int)) ) {
    System.out.println("Coordinates changed");
} }

cclass VerboseLogging extends Logging {
  after(): call(void Point.setColor(Color)) {
    System.out.println("Color changed");
} }

cclass Main {
  public static void main(String args[]) {
    Logging l = null;
    Point p[] = createSamplePoints();
    if (args[0].equals("-log")) l = new Logging();
    if (args[0].equals("-verbose")) l = new VerboseLogging();
    deploy (l) { modify(p); }
  } ...
}
```

The use of Logging or VerboseLogging is dynamic determined

Example: Dynamic Deployment

```
cclass Logging {
  after(): (call(void Point.setX(int)) || call(void Point.setY(int)) ) {
    System.out.println("Coordinates changed");
} }

cclass VerboseLogging extends Logging {
  after(): call(void Point.setColor(Color)) {
    System.out.println("Color changed");
} }

cclass Main {
  public static void main(String args[]) {
    Logging l = null;
    Point p[] = createSamplePoints();
    if (args[0].equals("-log")) l = new Logging();
    if (args[0].equals("-verbose")) l = new VerboseLogging();
    deploy (l) { modify(p); }
  } ...
}
```

Aspectual Polymorphism

CaesarJ Development Tool

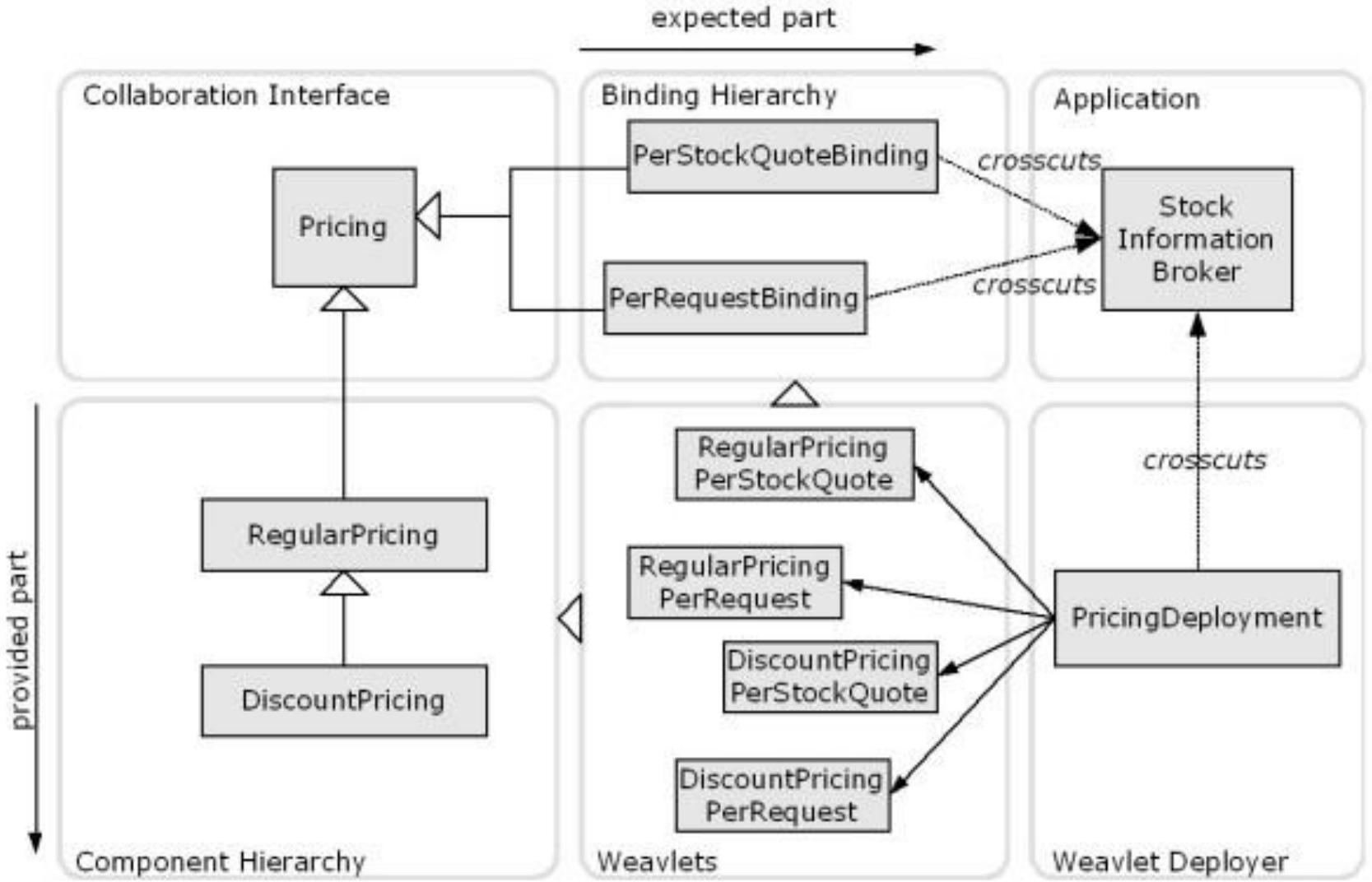
CJDT

- Provide IDE support for the CaesarJ
  - Plug-in for Eclipse
- Eclipse extensions
  - Editor with keyword highlighting
  - Outline view
  - New project wizard
  - CaesarJ hierarchy view
  - Debugging support

The Pricing Example

- Application for stock quotes
  - A server part
  - A client part
- The server includes a database to store information about quotes (*Hashtable*)
- The client interface provides a method to request stack quotes (*collectInfo*)

The Pricing Example



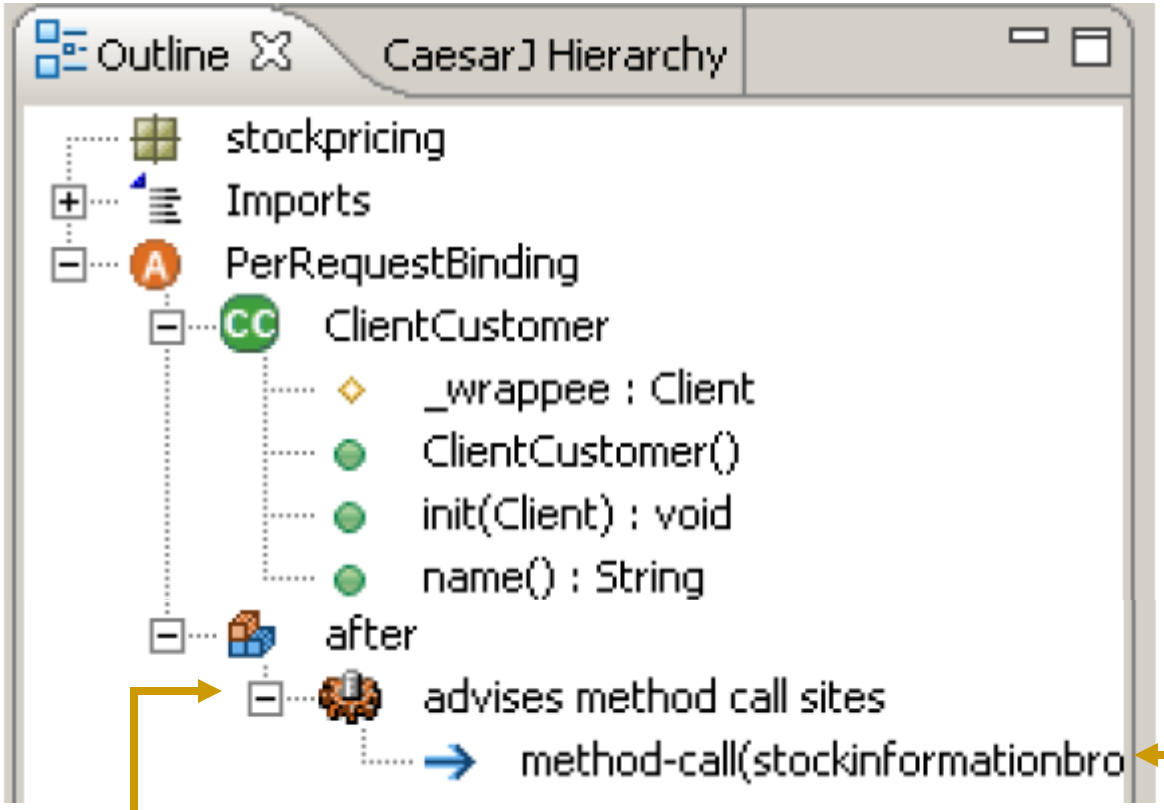
CaesarJ Editor

```
public deployed cclass PricingDeployment
{
    static Map pricingMapping = new HashMap();

    static
    {
        try
        {
            // User <-> Pricing Mapping
            pricingMapping.put("Klaus", new PerRequestDiscountPricing_Impl(null));
            pricingMapping.put("Egon", new PerStockQuoteDiscountPricing_Impl(null));
            pricingMapping.put("Mira", new PerRequestRegularPricing_Impl(null));
        }
        catch(Throwable t)
        {
            t.printStackTrace();
        }
    }

    void around(Client c) :
        (execution(void Client.run(String [])) && this(c))
    {
        deploy (pricingMapping.get(c.getName()))
        {
```

Deployed cclass weave crosscutting code

Around advice choose the pricing policy

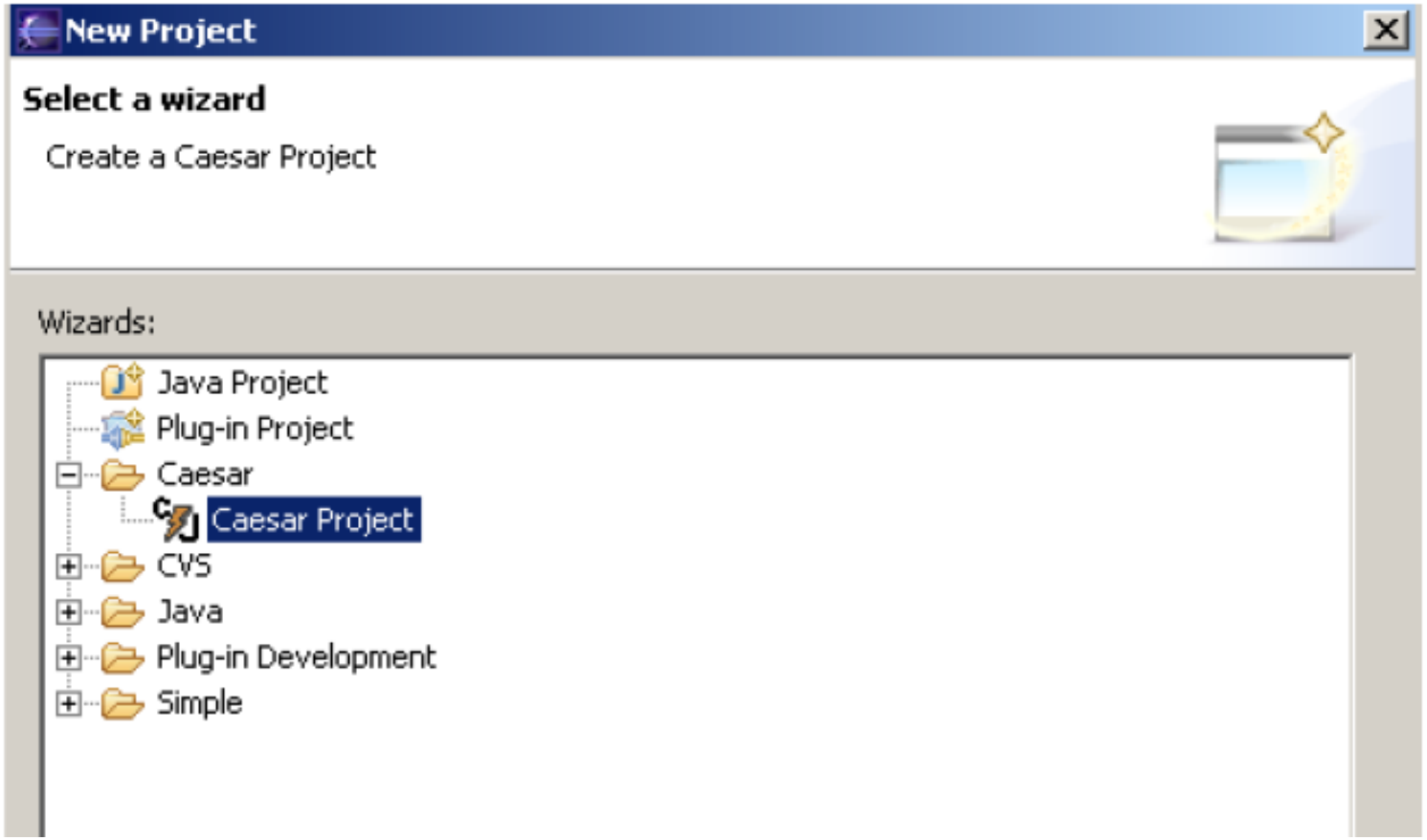
Outline view



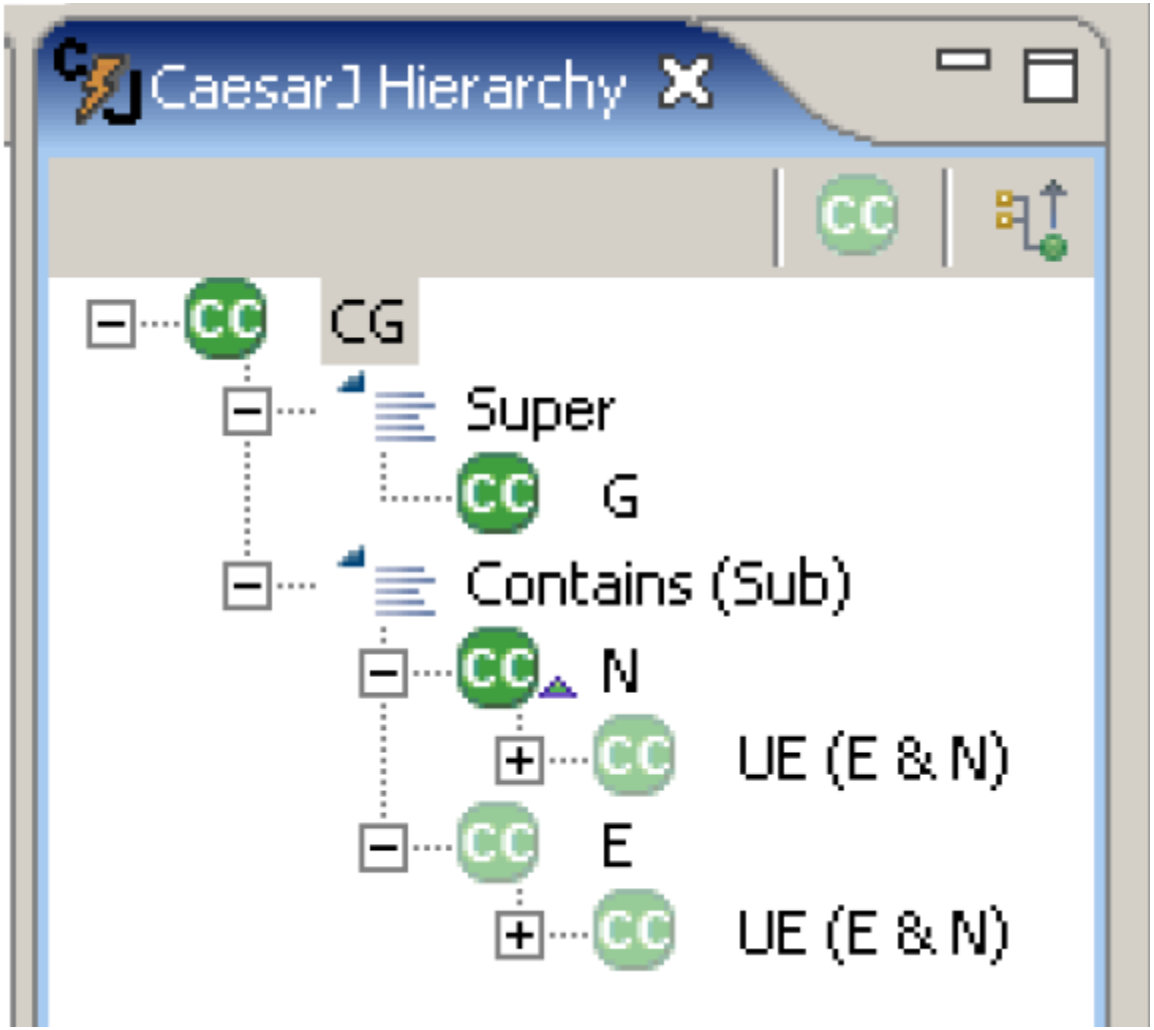
Advice declaration

Advised place

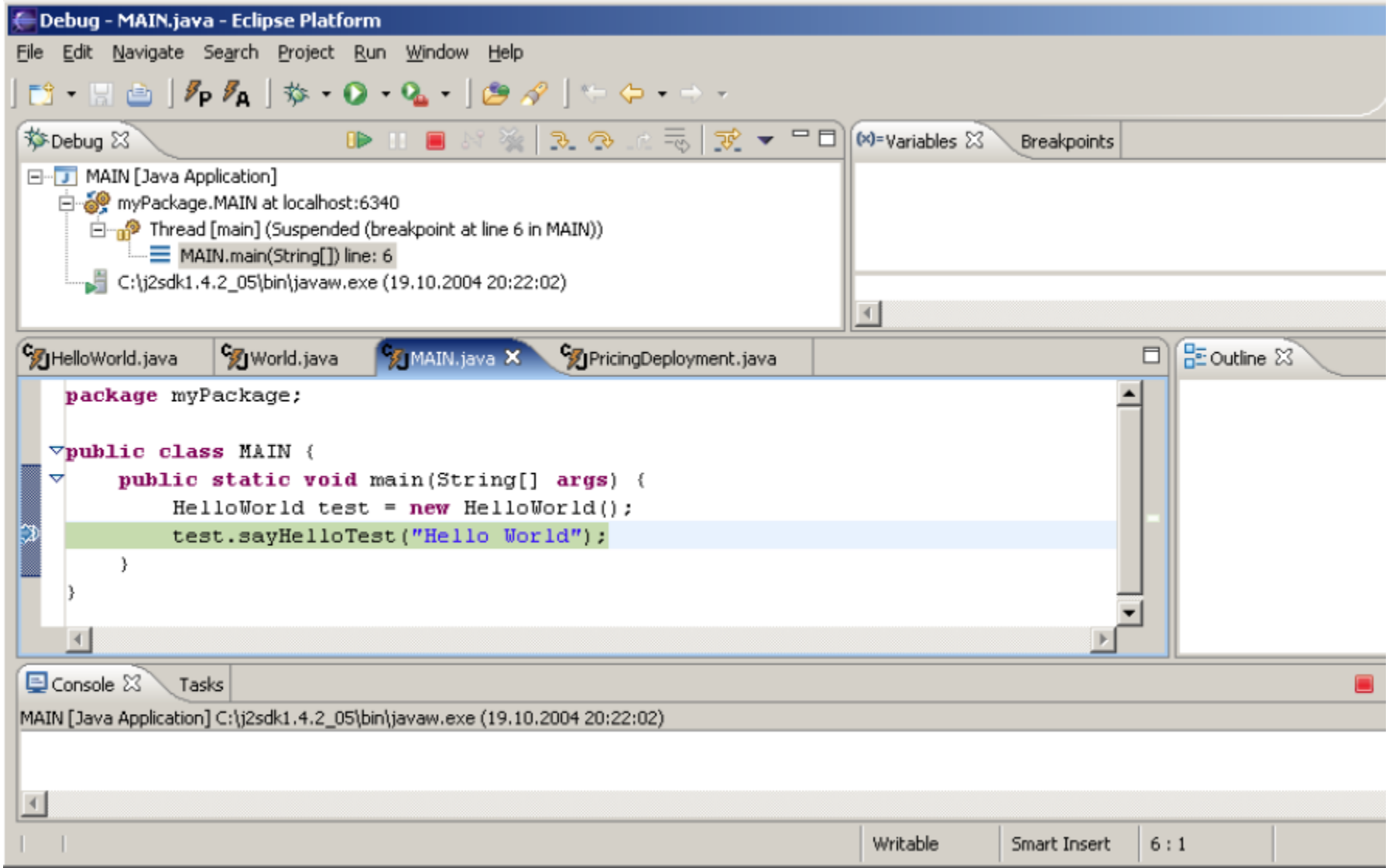
New CaesarJ Project



Hierarchy View



Debugging Support



Bibliography

- I. Aracic, V. Gasiunas, M. Mezini, and K.Ostermann. **Overview of CaesarJ**. Transactions on Aspect-Oriented Software Development I, vol. 3880, pp. 135-173, 2006.
- J. Unger and D. Zwicker. **User's Guide for CaesarJ Development Tool**. Version 0.4.0, 2004.
- http://www.caesarj.org