Engineering Software Development in Java

Lecture Notes for ENCE 688R, Civil Information Systems

Spring Semester, 2013

Mark Austin,
Department of Civil and Enviromental Engineering,
University of Maryland,
College Park,
Maryland 20742, U.S.A.

Copyright ©2012-2013 Mark A. Austin. All rights reserved. These notes may not be reproduced without expressed written permission of Mark Austin.

Contents

III Modeling System Structure **1**

12 Modeling Abstractions and Software Design Patterns **2**

12.1 Abstractions for Modeling System Structure 2
Components, Attributes and Relationships 2
Interconnect Design Methodology 3
Synthesis of Block Assemblies 4
Interface Contracts 5
12.2 Structural Design Patterns 6
Definition and Benefits 6
Summary of Structural Design Patterns 6
12.3 Bridge Design Pattern 9
Example 01. Working with Multiple APIs 10
Example 02. Customized Formatting of Lists 14
12.4 Adapter Design Pattern 20
Example 03. Adapter Interface for Incompatible Objects 20
Example 04. Table Adapter Model for Textual Requirements 25
12.5 Composite Design Pattern 31
Example 05. Composite Hierarchy for Furniture Placement 32
Example 06. Draw Composite Hierarchy of Furniture 37
12.6 Data-Flow Processing with Graphs of Executable Components 48
Example 07. Network of Arithmetic Block Processors 61
Example 08. Hierarchy of Components for Data-Flow Processing 70

References **73**

Part III

Modeling System Structure

Chapter 12

Modeling Abstractions and Software Design Patterns

12.1 Abstractions for Modeling System Structure

By definition, structure is an arrangement of parts, or constituent particles, in a substance or body (Merriam Webster, 1981). The system structure viewpoint focuses on the internal subsystems and how they are connected. This viewpoint helps us to understand how the system will accomplish its purpose? To understand structure at the meta-model level, we need to:

> **Identify the basic data types (or objects) that will exist within a domain and then devise mechanisms for describing connectivity and containment relations among these entities...**

Components, Attributes and Relationships

When an engineer describes the structure of a system, the description must contain a list of components/parts, attributes, and relationships:

1. **Components**

   Components are the operating parts of a system consisting of input, process, and output. Each system component may assume a variety of values to describe a system state.

2. **Attributes**

   Attributes are the properties of the components in the system. Most often the attributes characterize the system.

3. **Relationships**

   Relationships are the links between the components and attributes.

Most systems are assembled from a combination of components and subsystems. An important characteristic of a system is that its purpose is met by the properties of the system as "a whole," and not just by the union of the components.

For software, typical characteristics are data types associated with variables and methods. For systems hardware, characteristics can include things like size, weight, shape and location. In many of the traditional engineering disciplines, even in high-level representations of systems, geometric positioning – containment includes relationships like: Is object A inside object B? – and spatial layout of system objects is an important element of defining and evaluating system structure.

Interconnect Design Methodology

The interconnect design methodology assumes that complicated products and processes can be represented as hierarchies and networks of product and process blocks, i.e.,

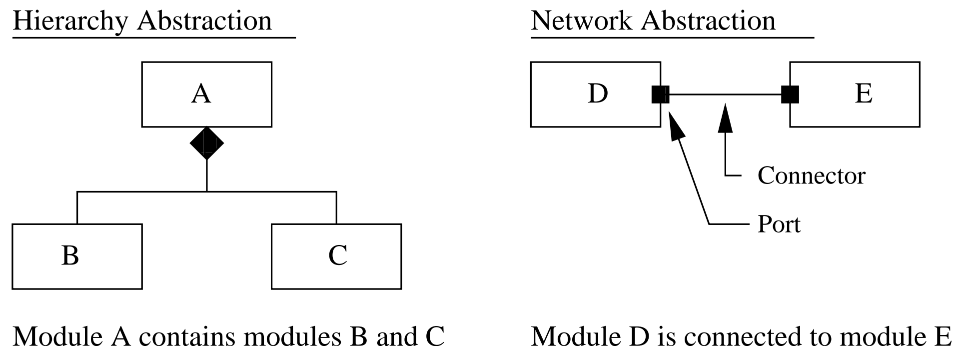


Figure 12.1. Hierarchy and network abstractions for SE development.

Assigning properties and functions to blocks is relatively straightforward.

Combining Hierarchy and Connectivity of Components. Figure 12.2 shows a simple system hierarchy decomposed into two levels, with a network of components (B and C) at level 2.

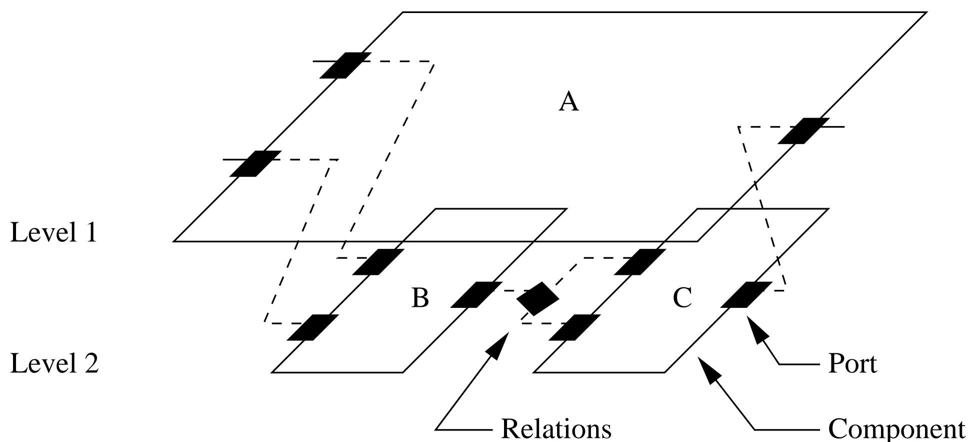


Figure 12.2. Hierarchy and network abstractions for SE development.

The system structure must satisfy the following constraints:

1. Within a hierarchy, each level is logically connected to the levels above and below it.

2. A port cannot be contained by more than one entity.

3. Links cannot cross levels in the hierarchy,

4. Port-to-port communications must have compatible data types (e.g., signal, energy, force).

Data values at lower levels aggregate into the data values at higher levels. Evaluation mechanisms should provide the designer with critical feedback on the feasibility and correctness of the system architecture.

System Assembly. Systems are assembled from parts, connectors, and ports with interface specifications:

1. Part
Parts represent a set of instances that are aggregated within a containing classifier instance. Parts may be joined by attached connectors, and specify configurations of linked instances.

2. Connector
Connectors specify a link (or an instance of an association) that enables communication between parts in a structure or with the surrounding environment.

3. Port
Each part attaches to other parts at specific locations. These locations are called ports. Ports specify an interaction point between a classifier and its environment and/or between a classifier and other internal ports.

4. Interface Specification
An interface specification defines precisely what a client of that interface/component can expect in terms of supplied operations and services (i.e., in terms of forces, material or energy or information protocols; minimum and maximum levels of component functionality), and operational pre- and post-conditions.

Synthesis of Block Assemblies

Creation of system architectures can be viewed via collections of modules and connectors, and rules for system assembly.

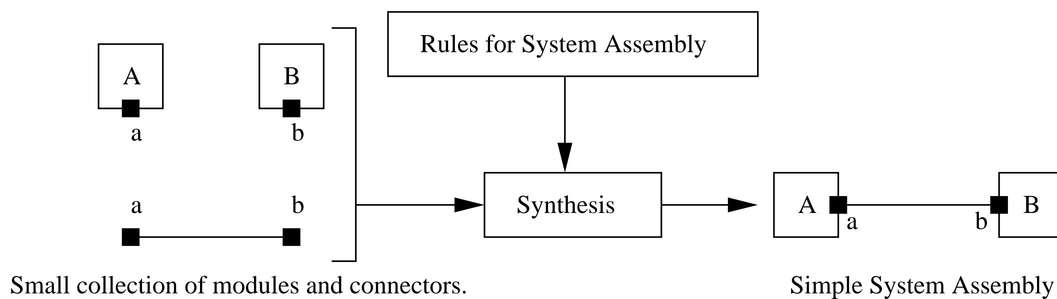


Figure 12.3. Synthesis of system architectures supported by product descriptions on Web.

More complicated sub-systems can be modeled graphs. Looking ahead, we need to be able to merge graph models and remove inconsistencies among viewpoints.

Interface Contracts

Complex objects have multiple interaction points (i.e., ports). Each port will be dedicated to a specific purpose and present and interface appropriate to that purpose – the pre- and post-conditions and satisfaction of the input requirements constitute a contract.

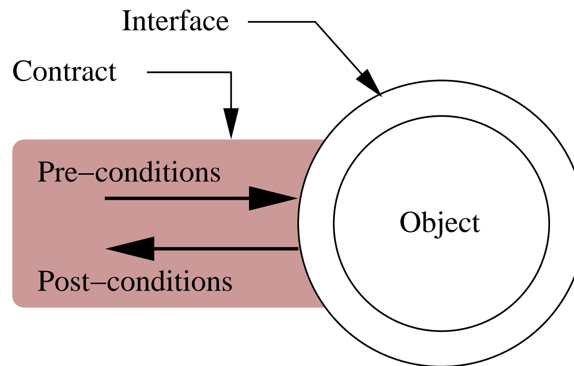


Figure 12.4. Role of pre- and post-conditions in contracts for object usage (Source: Newton A.R., "Notes on Interface-Based Design," EECS, UC Berkeley).

Networks of interacting/communicating objects having a larger purpose/functionality can be synthesized by stitching together objects that have compatibly I/O requirements and pre- and post-conditions.

12.2 Structural Design Patterns

Definition and Benefits

Definition. Experienced designers know that instead of always returning to first principles, routine design problems are best solved by adapting solutions to designs that have worked well for them in the past [1, 2]. A design pattern is simply a description of:

1. A problem that occurs over and over again, and
2. A core solution to that problem stated in such a way that it can be reused many times.

In other words, a design pattern prescribes a [problem, solution] pair. The design pattern identifies the participating subsystems and parts, their roles and collaborations, and distribution of responsibilities.

Design Patterns in Urban Design. For a wide range of domains, this approach to problem solving is popular because it encodes many years of professional experience in the how and why of design, and is time efficient.

Design patterns crop up in many avenues of day-to-day life. For example, as illustrated in Figures 12.5 and 12.6 the layout of streets in planned communities follows patterns, in part, because ...

> **... it is known that these patterns lead to efficiencies in the use of resources such as energy, etc. [1].**

Benefits. Gamma and co-workers [2] point out that patterns facilitate reuse – one persons pattern can be another persons fundamental building block. Software design patterns are particularly beneficial in the development of architectures for distributed systems (Buschmann, Henney, and Schmidt, 2007).

Summary of Structural Design Patterns

Structural design patterns provide ...

> **... effective ways to partition and to combine the elements of an application.**

Of course there are many ways that the elements of an application can be combined. This, in turn, results in an ensemble of structural design patterns.

1. **Adapter Design Pattern.**

   Acts as an intermediary between two classes, converting the interface of one class so that it can be used with the other.

2. **Bridge Design Pattern.**

   Divides a complex component into separate, but related, inheritance hierarchies. One hierarchy provides a functional abstraction; the second hierarchy provides the details of implementation.

   Organizing code in the way makes it easier to change either aspect of the component.

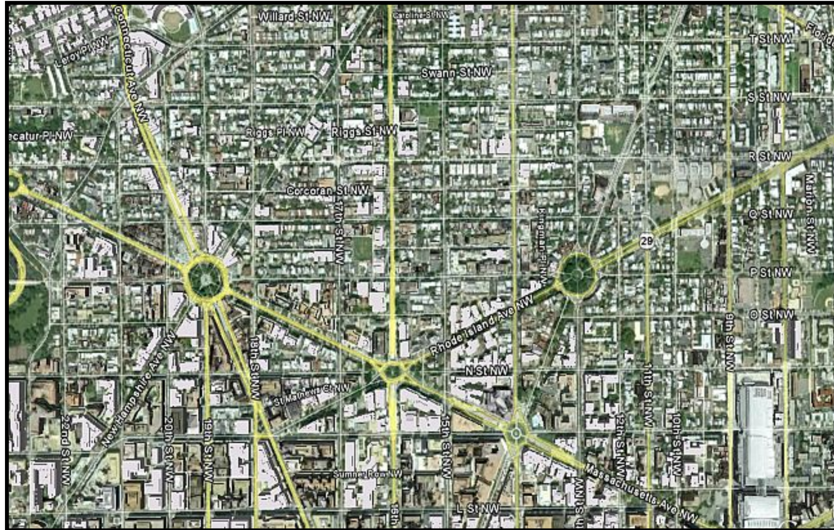


Figure 12.5. Urban design. Rectangular pattern with superimposed diagonals.

Figure 12.6. Urban design pattern. Suburbia.

3. **Composite Design Pattern.**

   This design pattern provides a flexible way to create hierarchical tree structures of arbitrary complexity, while enabling every element in the structure to operate with a uniform interface.

4. **Decorator Design Pattern.**

   This design pattern provides a way to flexibly add or remove component functionality without chaning its external appearance or function.

5. **Facade Design Pattern.**

   Provides a simplified interface to a group of subsystems or a complex subsystem.

6. **Proxy Design Pattern.**

   This design pattern provides a representative of another object, usually for reasons of functionality and performance, such as access, speed or security.

In this chapter we will focus on only three structural patterns: (1) The Bridge Design Pattern, (2) The Adapter Design Pattern, and (3) The Composite Design Pattern.

12.3 Bridge Design Pattern

The bridge pattern ...

> **... decouples an abstraction from its implementation, so that the two can vary independently.**

In many applications, the separation of abstraction from implementation can vastly simplify the management of code where classes and what they do vary often.

The class itself can be thought of as ...

> **... the implementation**

and what the class can do as ...

> **... the abstraction.**

In other words, the bridge pattern can also be thought of as two layers of abstraction.

A Simple Example. A household switch controlling lights, ceiling fans, etc. is an example of a bridge.

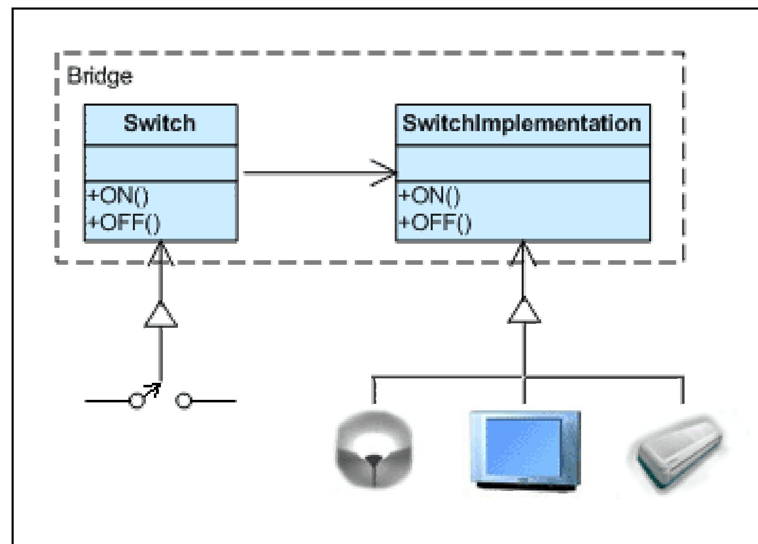


Figure 12.7. Schematic for the bridge design pattern.

The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.

Check list

1. Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.

2. Design the separation of concerns: what does the client want, and what do the platforms provide.

3. Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.

4. Define a derived class of that interface for each platform.

5. Create the abstraction base class that "has a" platform object and delegates the platform-oriented functionality to it.

6. Define specializations of the abstraction class if desired.

Example 01. Working with Multiple APIs

In this example, we develop code for drawing shapes through the use of multiple drawing application interfaces.

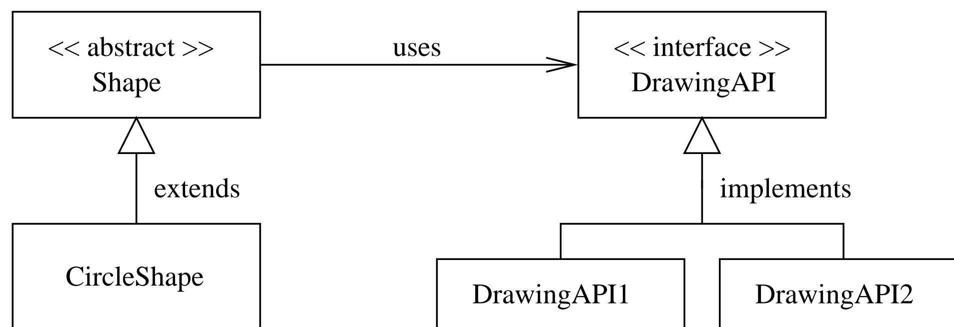


Figure 12.8. Class hierarchy for bridge to multiple drawing application interfaces.

As illustrated in Figure 12.8, CircleShape objects will be created as concrete extensions to an abstract Shape definition. Then, in turn, the abstract Shape class uses the DrawingAPI interface as a bridge to objects responsible for drawing circles (i.e., DrawingAPI1 and DrawingAPI2).

DrawingAPI.java. The DrawingAPI interface defines a formal abstraction against which concrete implementations can be programmed.

source code

```
/*
 *  ==========================================================
 *  DrawingAPI.java: Drawing API interface ...
 *  ==========================================================
 */

public interface DrawingAPI {
   public void drawCircle( double x, double y, double radius );
}
```

DrawingAPI1.java. Here is the first concrete implementation of a drawing application interface.

source code

```
/*
 *  ======================================================================
 *  DrawingAPI1: Concrete implementation of the DrawingAPI.
 *  ======================================================================
 */

public class DrawingAPI1 implements DrawingAPI {
   public void drawCircle(double x, double y, double radius) {
      System.out.printf("*** In DrawingAPI1(): ");
      System.out.printf("Circle at (%4.1f:%4.1f): Radius = %4.1f\n", x, y, radius);
   }
}
```

To keep things simple, we print a message and leave.

DrawingAPI2.java. And here is the second concrete implementation of a drawing application interface.

source code

```
/*
 *  ======================================================================
 *  DrawingAPI2: Concrete implementation of the DrawingAPI.
 *  ======================================================================
 */

public class DrawingAPI2 implements DrawingAPI {
   public void drawCircle(double x, double y, double radius) {
      System.out.printf("*** In DrawingAPI2(): ");
      System.out.printf("Circle at (%4.1f:%4.1f): Radius = %4.1f\n", x, y, radius);
   }
}
```

Shape.java. The drawing API is used by a class hierarchy of shapes.

source code

```
/*
 *  =================================================================
 *  Shape.java; Abstract definition of a shape ...
 *  =================================================================
 */

public abstract class Shape {
   protected DrawingAPI drawingAPI;

   protected Shape(DrawingAPI drawingAPI){
      this.drawingAPI = drawingAPI;
```

```
   }

   public abstract void draw();                            // low-level
   public abstract void resizeByPercentage(double pct);     // high-level
}
```

The root of this hierarchy is the abstract class Shape. Notice that Shape refers indirectly to concrete implentations of the drawing API through the variable declaration:

```
   protected DrawingAPI drawingAPI;
```

And although objects of types Shape will never be created, Shape contains a constructor method which will be called by sub-class constructor methods.

CircleShape.java. Circle shape is a concrete implementation of a shape.

source code

```
/*
 *  ======================================================================
 *  CircleShape.java: Create circle shapes with reference to DrawingAPI ...
 *  ======================================================================
 */

public class CircleShape extends Shape {
   private double x, y, radius;

   public CircleShape(double x, double y, double radius, DrawingAPI drawingAPI ) {
      super(drawingAPI);
      this.x      = x;
      this.y      = y;
      this.radius = radius;
   }

   // Low-level i.e. Implementation specific

   public void draw() {
        drawingAPI.drawCircle(x, y, radius);
   }

   // High-level i.e. Abstraction specific

   public void resizeByPercentage(double pct) {
        radius *= pct;
   }
}
```

RunBridgePattern.java. We exercise the bridge design pattern by creating three circle shape objects having a variety of drawing APIs.

source code

```
/*
 *  ===================================================================
 *  RunBridgePattern.java: Exercise methods in the bridge design pattern.
 *  ===================================================================
 */

public class RunBridgePattern {
   public static void main(String[] args) {

      // Create an array of circle shapes ....

      Shape s01 = new CircleShape( 1, 2,  3, new DrawingAPI1() );
      Shape s02 = new CircleShape( 5, 7,  5, new DrawingAPI2() );
      Shape s03 = new CircleShape( 1, 4, 11, new DrawingAPI1() );

      Shape[] shapes = new Shape[] { s01, s02, s03 };

      // Traverse array: resize and draw each shape item ...

      for (Shape shape : shapes) {
          shape.resizeByPercentage( 2.5 );
          shape.draw();
      }
   }
}
```

After compiling the test program in the usual way, the program output is:

```
prompt >> java RunBridgePattern
*** In DrawingAPI1(): Circle at ( 1.0: 2.0): Radius =  7.5
*** In DrawingAPI2(): Circle at ( 5.0: 7.0): Radius = 12.5
*** In DrawingAPI1(): Circle at ( 1.0: 4.0): Radius = 27.5
prompt >>
```

An alternative and potentially bettern implementation would separate the circle object creation from the specification of the drawing API. This would lead to code that is more dynamic – for example,

```
      Shape s01 = new CircleShape( 1, 2,  3, new DrawingAPI1() );
      Drawing dapi1 = new DrawingAPI1();
      Drawing dapi2 = new DrawingAPI2();

      s01.setDrawingAPI( dapi1 );
```

Then at a later point in the program execution you could change the drawing API, e.g.,

```
      s01.setDrawingAPI( dapi2 );
```

Example 02. Customized Formatting of Lists

This bridge design pattern example is adapted from the text of Stelting and Maassen [4]. It shows how the functionality for an ordered list implemenation can be extended through the use of a list interface and multiple abstraction classes.

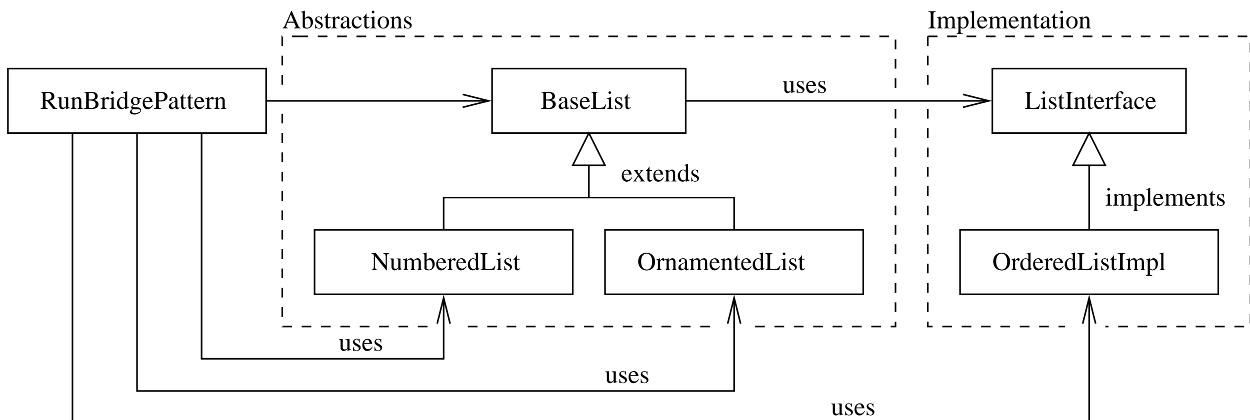


Figure 12.9. Class hierarchy plus annotations for an ordered list implementation that presents multiple abstractions to the outside world.

Figure 12.9 shows the relationship among the classes. The BaseList, NumberList and OrnamentedList classes present three different abstractions for an ordered list to outside test programs.

To see how this will work in practice, suppose that the list contains the string "One" as an item. The output for each abstraction might be:

```
One                    <-- BaseLine abstraction
1. One                 <-- Numerical abstraction
+  One                 <-- Ornamented abstraction
```

Each abstraction will communicate with the ordered list implemenation via the list interface.

BaseList.java. The baselist class provides general list capabilities.

source code

```
/*
 *  ===================================================================
 *  BaseList.java: Base List abstraction ...
 *  ===================================================================
 */

public class BaseList {
    protected ListInterface implementor;

    public void setImplementor( ListInterface impl ){
        implementor = impl;
    }
```

```
    public void add( String item ){
        implementor.addItem(item);
    }

    public void add(String item, int position ){
        if ( implementor.supportsOrdering() ) {
             implementor.addItem(item, position);
        }
    }

    public void remove(String item){
        implementor.removeItem(item);
    }

    public String get(int index){
        return implementor.getItem(index);
    }

    public int count(){
        return implementor.getNumberOfItems();
    }
}
```

Notice that when an operation is requested (e.g., add(..)) of the list, it is actually delegated across the bridge to the ListImplementation object.

NumberedList.java. This class creates a numbered list string representation for an item stored in the list.

source code

```
/*
 *  ===================================================================
 *  NumberedList.java: Return a string for a numbered list ...
 *  ===================================================================
 */

public class NumberedList extends BaseList {
    public String get(int index){
        return (index + 1) + ". " + super.get(index);
    }
}
```

OrnamentedList.java. An OrnamentedList expands on the BaseList model by adding a list character.

source code

```
/*
 *  ===================================================================
```

```
 *  OrnamentedList.java: Generate ornamented string representation.
 *  =================================================================
 */

public class OrnamentedList extends BaseList {
    private char itemType;

    public char getItemType(){ return itemType; }
    public void setItemType( char newItemType ){
        if (newItemType > ' '){
            itemType = newItemType;
        }
    }

    public String get(int index){
        return itemType + " " + super.get(index);
    }
}
```

ListInterface.java. The list interface defines the methods that an implementation will need to provide.

source code

```
/*
 *  =================================================================
 *  ListInterface.java: Interface to various implementations of a List
 *  =================================================================
 */

public interface ListInterface {
   public void    addItem(String item);
   public void    addItem(String item, int position);
   public void    removeItem(String item);
   public int     getNumberOfItems();
   public String  getItem(int index);
   public boolean supportsOrdering();
}
```

OrderedListImpl.java. This class provides the underlying storage capability for the list, and can be flexibly paired with the three classes which provide the abstraction.

source code

```
/*
 *  =================================================================
 *  OrderedListImpl.java: Implementation for an Ordered List...
 *  =================================================================
 */

import java.util.ArrayList;
```

```
public class OrderedListImpl implements ListInterface {

    private ArrayList items = new ArrayList();

    public void addItem(String item){
        if (!items.contains(item)){
            items.add(item);
        }
    }
    public void addItem(String item, int position){
        if (!items.contains(item)){
            items.add(position, item);
        }
    }

    public void removeItem(String item){
        if (items.contains(item)){
            items.remove(items.indexOf(item));
        }
    }

    public boolean supportsOrdering(){
        return true;
    }

    public int getNumberOfItems(){
        return items.size();
    }

    public String getItem(int index){
        if (index < items.size()){
            return (String)items.get(index);
        }
        return null;
    }
}
```

In this particular implementation, the list items are stored in an ArrayList.

RunBridgePattern.java. Our test program assembles a BaseList containing four String items – One, Two, Three and Four. It then passes a copy of the implementation reference to instances of OrnamentedList and NumberedList. A this point there is one list in memory with three refences to it. Finally, the test program prints the list's contents in each of three abstraction styles.

source code

```
/*
 *  ============================================================
 *  RunBridgePattern.java: Exercise methods in bridge pattern...
 *  ============================================================
 */
```

```
public class RunBridgePattern {

    public static void main(String [] arguments){

        System.out.println("============================");
        System.out.println("Run Bridge pattern          ");
        System.out.println("============================");

        // Create ordered list object ...

        ListInterface implementation = new OrderedListImpl();

        // Create baseline object ...

        BaseList listOne = new BaseList();

        // Assign ordered list implementation to base ...

        listOne.setImplementor(implementation);

        // Now add elements to the list ...

        listOne.add("One");
        listOne.add("Two");
        listOne.add("Three");
        listOne.add("Four");

        // Create an ornamented list object ...

        OrnamentedList listTwo = new OrnamentedList();
        listTwo.setImplementor(implementation);
        listTwo.setItemType('+');

        // Create an numbered list object ...

        NumberedList listThree = new NumberedList();
        listThree.setImplementor(implementation);
        System.out.println();

        // Print contents of the base list ...

        System.out.println("Printing out first list (BaseList)");
        for (int i = 0; i < listOne.count(); i++){
            System.out.println("\t" + listOne.get(i));
        }
        System.out.println();

        // Print contents of the ornamented list ...

        System.out.println("Printing out second list (OrnamentedList)");
        for (int i = 0; i < listTwo.count(); i++){
            System.out.println("\t" + listTwo.get(i));
        }
        System.out.println();
```

```
        // Print contents of the numbered list ...

        System.out.println("Printing our third list (NumberedList)");
        for (int i = 0; i < listThree.count(); i++){
            System.out.println("\t" + listThree.get(i));
        }
    }
}
```

The program output is as follows:

```
prompt >>
prompt >> java RunBridgePattern
=============================
Run Bridge pattern
=============================

Printing out first list (BaseList)
One
Two
Three
Four

Printing out second list (OrnamentedList)
+ One
+ Two
+ Three
+ Four

Printing our third list (NumberedList)
1. One
2. Two
3. Three
4. Four
prompt >>
```

A Few Remarks

1. Adapters makes things work after they are designed. Bridges makes them work beforehand.
2. Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.
3. Adapter is meant to change the interface of an existing object.
4. State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the handle/body idiom. They differ in intent - that is, they solve different problems.

12.4 Adapter Design Pattern

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

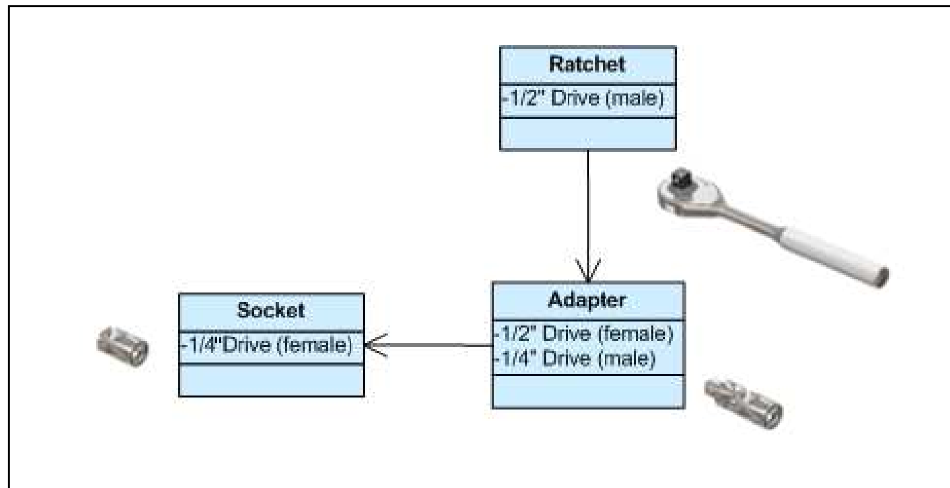


Figure 12.10. Socket metaphor for implementation of the adapter design pattern.

Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2 and 1/4. Obviously, a 1/2 drive ratchet will not fit into a 1/4 drive socket unless an adapter is used. A 1/2 to 1/4 adapter has a 1/2 female connection to fit on the 1/2 drive ratchet, and a 1/4 male connection to fit in the 1/4 drive socket.

Source: http://sourcemaking.com/design_patterns/adapter

Example 03. Adapter Interface for Incompatible Objects

In this example we will develop an adapter interface for dealing with the drawing of line and rectangular objects in a completely uniform way.

Code Before Adapter: Source: http://sourcemaking.com/design_patterns/adapter

Suppose that in some legacy code, lines are defined by pairs of end points (e.g., (x_1, y_1) and (x_2, y_2)), and rectangles are defined by a lower left-hand (x,y) coordinate point, plus dimensions for the rectangle width and height. Routines for drawing lines and rectangles might look like:

```
public class LegacyLine {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("line from (" + x1 + ',' + y1 + ") to (" + x2 + ',' + y2 + ')');
    }
}
```

and

```
public class LegacyRectangle {
    public void draw(int x, int y, int w, int h) {
        System.out.println("rectangle at (" + x + ',' + y + ") with width " + w
          + " and height " + h);
    }
}
```

Because the interface between Line and Rectangle objects is incapatible, the user has to recover the type of each shape and manually supply the correct arguments, i.e.,

source code

```
public class AdapterDemo {
    public static void main(String[] args) {

        // Create and array of legacy shapes ...

        Object[] shapes = { new LegacyLine(), new LegacyRectangle() };

        // A begin and end point from a graphical editor

        int x1 = 10, y1 = 20, x2 = 30, y2 = 60;

        for (int i = 0; i < shapes.length; ++i)
          if (shapes[i].getClass().getName().equals("LegacyLine") == true)
              ((LegacyLine)shapes[i]).draw(x1, y1, x2, y2);
          else if (shapes[i].getClass().getName().equals("LegacyRectangle") == true)
              ((LegacyRectangle)shapes[i]).draw(Math.min(x1, x2), Math.min(y1, y2)
                  , Math.abs(x2 - x1), Math.abs(y2 - y1));
    }
}
```

In our bare-bones implementation, we decide which version of the draw() method should be called by directly relating it to the class name, e.g.,

```
if ( shapes[i].getClass().getName().equals("LegacyLine") == true ) ....
```

While this approach to implementation will work, the lack of separation in the abstraction and implementation means that the code may need to be changed in numerous places if a new shape is added (e.g., LegacyTriangle.java, LegacySquare.java).

Code After Adapter

The adapter introduces and additional level of indirection to create a common interface mapping to legacy-specific peculiar interfaces. This results in the class hierarchy relationship shown in Figure 12.11.

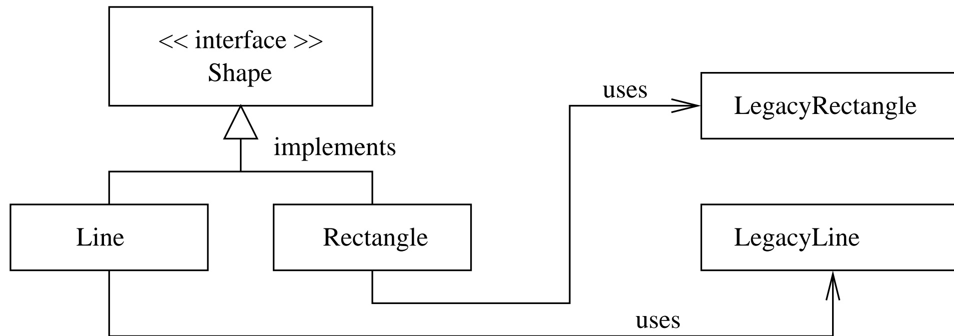


Figure 12.11. Class hierarchy for the adapter interfaces to legacy code.

The revised source code is as follows:

LegacyLine.java

source code

```
/*
 *  ==========================================================
 *  LegacyLine.java:
 *  ==========================================================
 */

public class LegacyLine {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.printf("Line from ( %2d, %2d ) to ( %2d, %2d )\n",
                          x1, y1, x2, y2 );
    }
}
```

LegacyRectangle.java

source code

```
/*
 *  ==========================================================
 *  LegacyRectangle.java:
 *  ==========================================================
 */

public class LegacyRectangle {
   public void draw(int x, int y, int w, int h) {
      System.out.printf("Rectangle at ( %2d, %2d ): width = %2d : height = %2d\n",
```

```
                        x, y, w, h );
   }
}
```

Shape.java

source code

```
/*
 *  ==========================================================
 *  Shape.java:
 *  ==========================================================
 */

public interface Shape {
   void draw(int x1, int y1, int x2, int y2);
}
```

Line.java

source code

```
/*
 *  ==========================================================
 *  Line.java: Adapter object to legacy line ...
 *  ==========================================================
 */

public class Line implements Shape {
   private LegacyLine adaptee = new LegacyLine();

   public void draw(int x1, int y1, int x2, int y2) {
       adaptee.draw(x1, y1, x2, y2);
   }
}
```

Rectangle.java

source code

```
/*
 *  ==========================================================
 *  Rectangle.java: Adapter to legacy rectangle format.
 *  ==========================================================
 */

public class Rectangle implements Shape {
```

```
    private LegacyRectangle adaptee = new LegacyRectangle();

    public void draw(int x1, int y1, int x2, int y2) {
        int width  = Math.abs(x2 - x1);
        int height = Math.abs(y2 - y1);

        adaptee.draw(Math.min(x1, x2), Math.min(y1, y2), width, height );
    }
}
```

RunAdapterPattern.java. The test program creates an array of references to shape objects, one line and one rectangle. Then instances of Line and Rectangle, refer indirectly to implementations for the legacy line and rectangle.

source code

```
/*
 *  ==========================================================
 *  RunAdapterPattern.java:
 *  ==========================================================
 */

public class RunAdapterPattern {
    public static void main(String[] args) {

        // Create and array of references to shapes ...

        Shape[] shapes = { new Line(), new Rectangle() };

        // Call routine to draw individual shapes ...

        int x1 = 10, y1 = 20;
        int x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i)
            shapes[i].draw(x1, y1, x2, y2);
    }
}
```

The program output is as follows:

```
prompt >> java RunAdapterPattern
Line from ( 10, 20 ) to ( 30, 60 )
Rectangle at ( 10, 20 ): width = 20 : height = 40
prompt >>
```

Example 04. Table Adapter Model for Textual Requirements

In this example we develop an adapter to map an array of requirements objects into a simple requirements table view. See Figure 12.12.

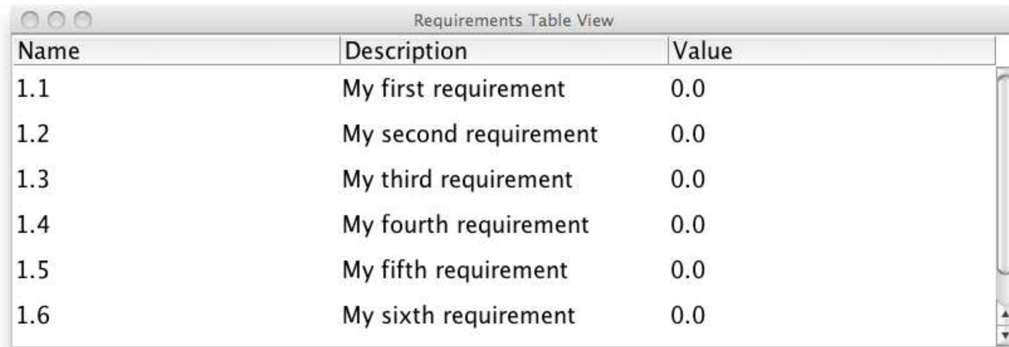


Figure 12.12. Screendump of a simple requirements table view.

Figure 12.13 shows the essential details of the class hierarchy needed to create the array-to-table adapter model. The abstract table model defines methods for accessing information about the table size, the contents within a particular cell, and the machinery for listening to and handling table events.

The abbreviated details of AbstractTableModel are as follows:

```
public abstract class AbstractTableModel implements TableModel, Serializable {
   public String getColumnName(int columnIndex) { ... }
   public int findColumn(String columnName) { ... }
   public void setValueAt(Object value, int rowIndex, int columnIndex) { ..  }
   public abstract Object getValueAt(int row, int column);
   public abstract int getColumnCount();
   public abstract int getRowCount();
}
```

The TableModel interface specifies methods that need to be implemented in order for JTable to operate. Again, the abbreviate details are:

```
import javax.swing.event.TableModelListener;

public interface TableModel {
   public int getRowCount();
   public int getColumnCount();
   public String getColumnName(int columnIndex);
   public Class getColumnClass(int columnIndex);
   public boolean isCellEditable(int rowIndex, int columnIndex);
   public Object getValueAt(int rowIndex, int columnIndex);
   public void setValueAt(Object aValue, int rowIndex, int columnIndex);
   public void addTableModelListener(TableModelListener listener);
   public void removeTableModelListener(TableModelListener listener);
}
```

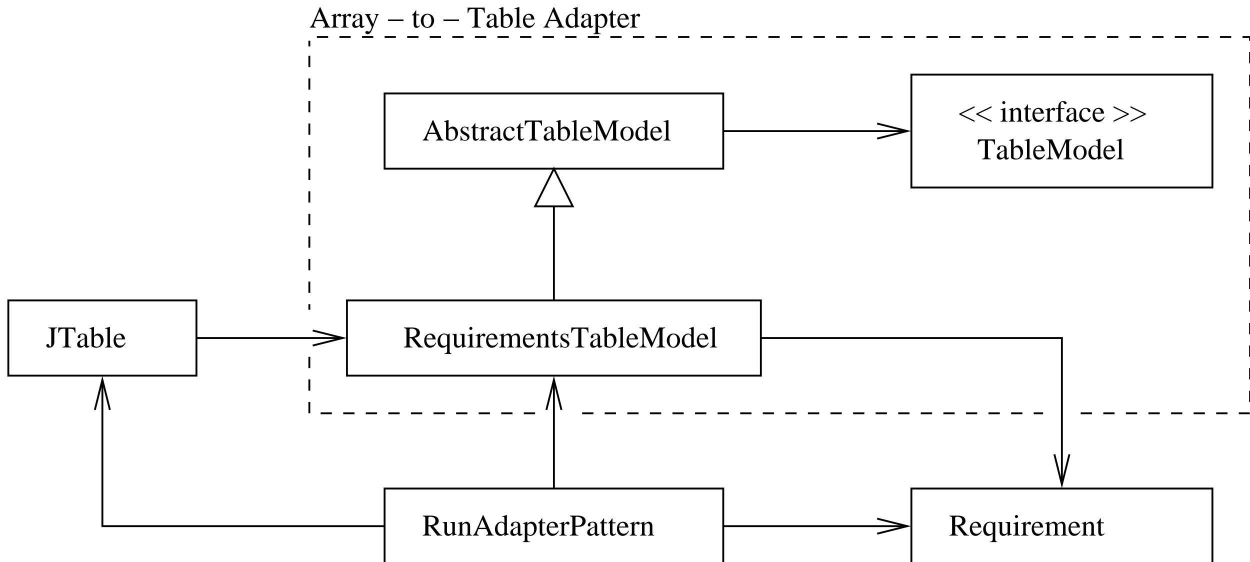


Figure 12.13. Class hierarchy for requirements table program.

Where it makes sense, default implementations for these methods can be provided in AbstractTableModel. Implementations for the remaining methods are provided by an adapter class that extends AbstractTableModel, i.e., in our application, this is RequirementsTableModel.

The beauty of this setup is that it allows JTable to refer to adapter models simply in terms of the methods prescribed in the TableModel interface – JTable never actually needs to know about the working details of RequirementsTableModel.

Requirement.java. This class defines methods and attributes for a single textual requirement and its value.

source code

```
/*
 *  ==================================================================
 *  Requirement.java: This class defines a single textual requirement.
 *
 *  Written by: Mark Austin                                 March 2012
 *  ==================================================================
 */

import java.lang.Math.*;

public class Requirement {
    String sName;
    String sDescription;
    double dValue;

    // Constructor

    public Requirement() {}
```

```
public Requirement( String sName, String sDescription ) {
    this.sName = sName;
    this.sDescription = sDescription;
    this.dValue = 0.0;
}

// Set and get requirement name ...

public void setName( String sName ) {
    this.sName = sName;
}

public String getName() {
    return this.sName;
}

// Set and get requirement description ...

public void setDescription( String sDescription ) {
    this.sDescription = sDescription;
}

public String getDescription() {
    return this.sDescription;
}

// Set and get value of requirement ...

public void setValue( double dValue ) {
    this.dValue = dValue;
}

public double getValue() {
    return this.dValue;
}

// Copy Requirement parameters to a string format ...

public String toString() {
   return "Req(" + sName + ") is (" + sDescription + "): value = " + dValue;
}

// Exercise methods in class Circle ...

public static void main( String [] args ) {

   System.out.println("Exercise methods in class Requirement");
   System.out.println("=====================================");

   Requirement rA = new Requirement();
   rA.setName ("1.0");
   rA.setDescription("My first requirement");
   rA.setValue( 3.0 );

   System.out.println("Requirement" );
```

```
        System.out.println( rA );
    }
}
```

As a standalone program, this source code file generates the output:

```
prompt >> java Requirement
Exercise methods in class Requirement
=====================================
Requirement
Req(1.0) is (My first requirement): value = 3.0
prompt >>
```

RequirementsTableModel.java. This is the interesting part of the array-to-table adapter. While AbstractTableModel talks about a table model in generic terms, RequirementsTableModel defines the size and contents of a table having three columns.

source code

```
/*
 * ====================================================================
 * RequirementsTableModel.java: Adapt a collection of requirements for
 *                              display in a JTable format.
 *
 * Written By: Mark Austin                                   March 2012
 * ====================================================================
 */

import javax.swing.table.*;

public class RequirementsTableModel extends AbstractTableModel {

    protected Requirement[] requirements;
    protected String[] columnNames = new String[] { "Name", "Description", "Value" };

    // Construct a table of requirements from an array of requirements.

    public RequirementsTableModel( Requirement[] requirements ) {
        this.requirements = requirements;
    }

    // Return the number of columns in this table.

    public int getColumnCount() {
        return columnNames.length;
    }

    // Return the name of the indicated column.

    public String getColumnName(int i) {
        return columnNames[i];
    }
```

```
    // Return the number of rows in this table.

    public int getRowCount() {
        return requirements.length;
    }

    // Return the value at the indicated row and column..

    public Object getValueAt(int row, int col) {
        switch (col) {
        case 0:
             return requirements[row].getName();
        case 1:
             return requirements[row].getDescription();
        case 2:
             return new Double( requirements[row].getValue() );
        default:
             return null;
        }
    }
}
```

The requirements adapter stores the requirements as an array of references to individual requirements. The methods getRowCount(..) and getValueAt(..) retrieve the contents of the requirements matrix.

Notice that we do not define methods for the event listeners – these are defined in AbstractTableModel and inherited through the extension mechanism.

RunAdapterPattern.java. The test program instantiates seven textual requirement objects, organizes their referenes into an array called `ra`, creates and object for the requirments table model, and finally, creates and displays a JTable within a frame.

source code

```
/*
 * ============================================================================
 * RunAdapterPattern.java: Use a RequirementsTableModel adapter interface to
 *                         link individual requirements to a JTable view.
 *
 * Written By: Mark Austin                                          March 2012
 * ============================================================================
 */

import java.awt.Component;
import java.awt.Font;

import javax.swing.*;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
```

```
public class RunAdapterPattern {
   public static void main(String[] args) {

      // Initialize user interface fonts ....

      Font font = new Font("Dialog", Font.PLAIN, 18);
      UIManager.put("Table.font", font);
      UIManager.put("TableHeader.font", font);

      // Generate a small set of requirements ...

      Requirement r1 = new Requirement("1.1", "My first requirement" );
      Requirement r2 = new Requirement("1.2", "My second requirement" );
      Requirement r3 = new Requirement("1.3", "My third requirement" );
      Requirement r4 = new Requirement("1.4", "My fourth requirement" );
      Requirement r5 = new Requirement("1.5", "My fifth requirement" );
      Requirement r6 = new Requirement("1.6", "My sixth requirement" );
      Requirement r7 = new Requirement("1.7", "My seventh requirement" );

      // Create a requirements table model ...

      Requirement ra [] = { r1, r2, r3, r4, r5, r6, r7 };
      RequirementsTableModel rtm = new RequirementsTableModel( ra );

      // Create and view table model.

      JTable table = new JTable( rtm );
      table.setRowHeight(36);

      JScrollPane pane = new JScrollPane(table);
      pane.setPreferredSize( new java.awt.Dimension(800, 250) );

      // Create and display frame ....

      JFrame frame = new JFrame("Requirements Table View");
      frame.getContentPane().add( pane );
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.pack();
      frame.setVisible(true);
   }
}
```

12.5 Composite Design Pattern

Purpose. To develop a flexible way to create hierarchical tree structures of arbitrary complexity while enabling every element of the structure to operate with a uniform interface.

Description. Composite allows a group of objects to be treated in the same way as a single instance of an object. The intent of composite is to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions uniformly.

Implementation. Implementations of this pattern employ component, node, and composite classes:

1. The base component provides the core model, defining all methods and/or variables to be used by all objects in the class Composite.
2. The node (or leaf) classes represent terminal behavior (i.e., parts of the composite that cannot contain other components).
3. Composite (or branch) classes can have other components added to them, thereby permitting extension of the Composite structure.

Figure 12.14 shows the class diagram for the composite design pattern.

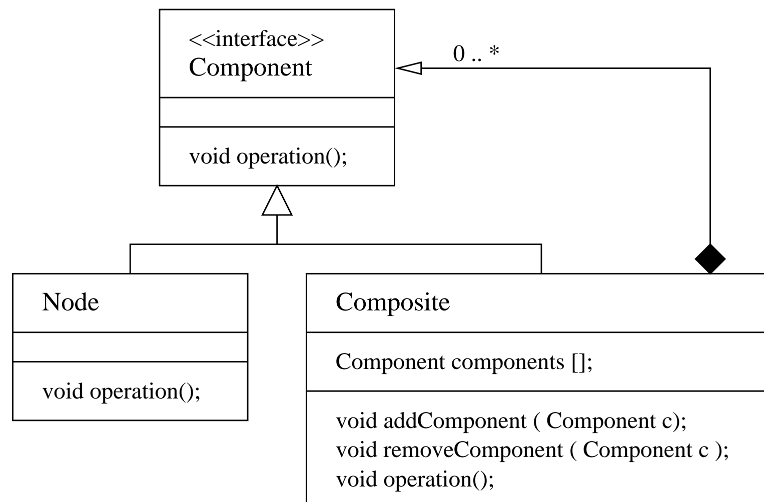


Figure 12.14. Composite class diagram (pg. 159 of Stelting).

Implementation of the composite design pattern requires three elements:

1. **Component.** The component interface defines methods available to all parts of the tree structure. Normally component is implemented as an abstract class. Its subclass (Node) is a concrete class and is used to create collections or a tree structure.
2. **Composite.** This class is defined by the components it contains. It supports a dynamic group of Components, and so it has methods to add and remove Component objects from its collection.

The Composite class also provides concrete implementions of operational methods defined in class Component.

3. **Node.** The Node class implements the Component interface and provides an implementation for each of the component's operational methods.

Notice that unlike the Composite, Node is a leaf – it does not contain any references to other components.

Example 05. Composite Hierarchy for Furniture Placement

Within buildings, rooms are spaces with an assigned function and a given set of characteristics. Within rooms, collections of furniture provide support for required functions. For example, an office is a space dedicated to work. A kitchen is a space dedicated to the preparation of food. Each item within a room, whether it be a desk or chair of benchtop for preparation of food, occupies space. Collectively, the building-room-item decomposition constitutes a nested hierarchy of spaces.

A second insight from architectural design is that collections of items work together to support a functional need. For example, an office is not an office without a desk. And a desk is of little use unless there is a chair positioned in a way that allows for usage of the desk space. The natural way of encoding these needs is to say [3]:

> **... spaces can include one or more sub-spaces (children), and are governed by rules for how things should be configured.**

To see how these ideas might be implemented in practice, Figures 12.15 and 12.16 show plan- and tree-views for the nesting and placement of spaces and furniture in a simple office. Notice that in the office definition, we first define a workspace, and then position furniture within the workspace. Each space in the hierarchy can be augmented with information on its position, default size, range of permissible values, an abstractions for visualizing the space/artifact. For example, while the geometry of furniture can be complex, the shape of a given piece can be approximated by a tightly fitting bounding box.

Abbreviated Coding the Composite Hierarchy

Let's now look at the structure of a composite hiearchy implementation for the workspace and furniture layout. Using the composite design pattern, we can draw or print the details of space and furniture placement by recursively visiting spaces, starting at a given room (root space).

To keep things simple, we will print a string description of the furniture items and keep track of the layer level. Book keeping details of the coordinate transformations and rotations will be added in the next example.

Here is a summary of the required functionality for this example:

1. Office and Workspace contain children and, as such, can be viewed as containers in the hierarchy. Will will simply print the layer level.

2. Desk and Chair a leaf nodes in the hierarchy – they do not have children. We will print the name of the furniture item.

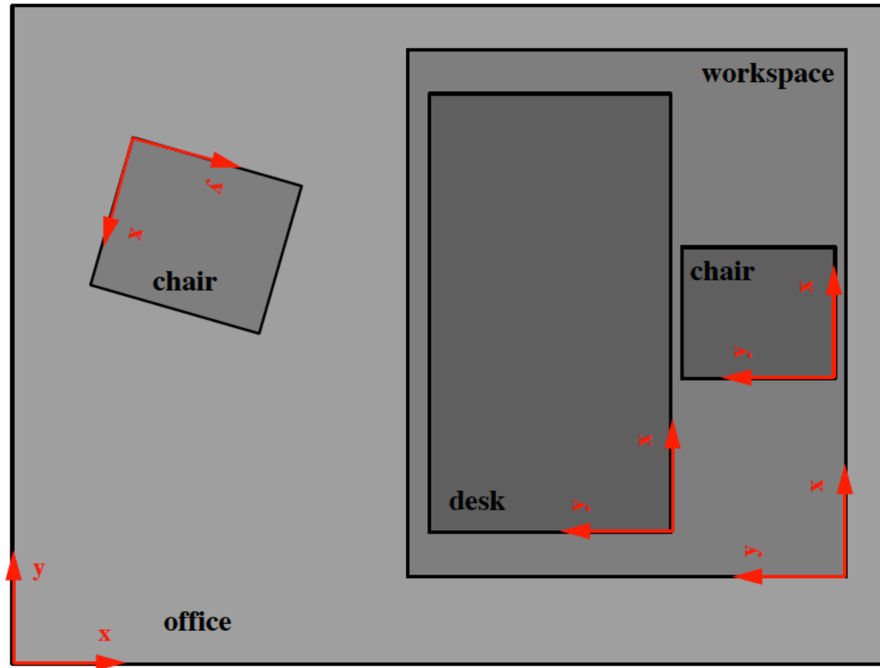


Figure 12.15. Composite hierarchy for layout of spaces and components (Source: [3]).

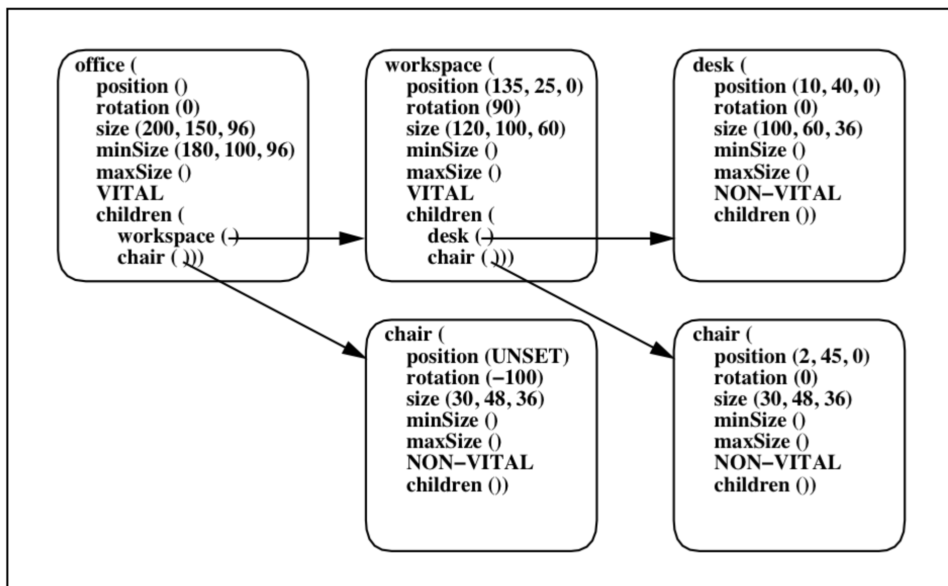


Figure 12.16. Tree hierarchy for layout of furniture in a simple office (Source: [3]).

The composite hiearchy design pattern provides a uniform view of containers and leaf nodes through the PrintView interface, i.e.,

```
    public interface PrintView {
       public void print();
    }
```

A stripped down version of the Chair object might look like:

```
/*
 *  ===================================================
 *  Chair.java: Code to create and print chair objects.
 *  ===================================================
 */

public class Chair implements PrintView {
    String sName; // Name of chair ...

    public Chair ( String sName ) {
       this.sName  = sName;
    }

    public void print() {
       System.out.println("*** Chair: " + sName );
    }
}
```

Table objects will follow a similar format. Now we can define the CompositeHierarchy for systematically traversing the tree structure of containers and leaf components, i.e.,

```
/*
 *  ================================================================
 *  CompositeHierarchy.java: Hierarchy of composite/furniture items.
 *  ================================================================
 */

import java.util.ArrayList;

public class CompositeHierarchy implements PrintView {
    private static int iCurrentLevel = 0;
    private String sName;

    // Collection of child printitems.

    private ArrayList<PrintView> children = new ArrayList<PrintView>();

    // Constructor for composite hierarchy ...

    public CompositeHierarchy( String sName ) {
       this.sName = sName;
    }

    // Print the composite graphic.
```

```
    public void print() {

       // Increment level ...

       iCurrentLevel = iCurrentLevel + 1;
       System.out.printf("*** Start level %1d : %s\n", iCurrentLevel, sName );

       // Loop to print the children items ....

       for (PrintView printview : children) {
             printview.print();
       }

       // Decrement level ...

       System.out.printf("*** Finish level %1d : %s\n", iCurrentLevel, sName );
       iCurrentLevel = iCurrentLevel - 1;
    }

    // Add composite/furniture item to the composition.

    public void add( PrintView printitem ) {
        children.add( printitem );
    }

    // Remove composite/furniture item the composition.

    public void remove( PrintView printitem ) {
        children.remove( printitem );
    }
}
```

Test Program. The test program creates composite objects for the office and workspaces, chair and table objects, and then systematically assembles the composite hierarchy.

```
/*
 * ==========================================================================
 * TestComposite.java: Create and print hierarchy of spaces and furniture items.
 * ==========================================================================
 */

public class TestComposite {
   public static void main(String[] args) {

      // Initialize three composite graphics

      CompositeHierarchy office    = new CompositeHierarchy("Office");
      CompositeHierarchy workspace = new CompositeHierarchy("Workspace");

      // Create table and chairs ...

      Table    desk = new Table("Office Desk");
      Chair chair01 = new Chair("Office Chair");
```

```
      Chair chair02 = new Chair("Customer Chair");

      // Add chair and table to the workspace ...

      workspace.add( chair01 );
      workspace.add( desk );

      // Add customer chair and workspace to the office space ...

      office.add( chair02 );
      office.add( workspace );

      // Traverse composite hierarchy ...

      office.print();
   }
}
```

systematically assembles the hierarchy of spaces and furniture shown in Figure .

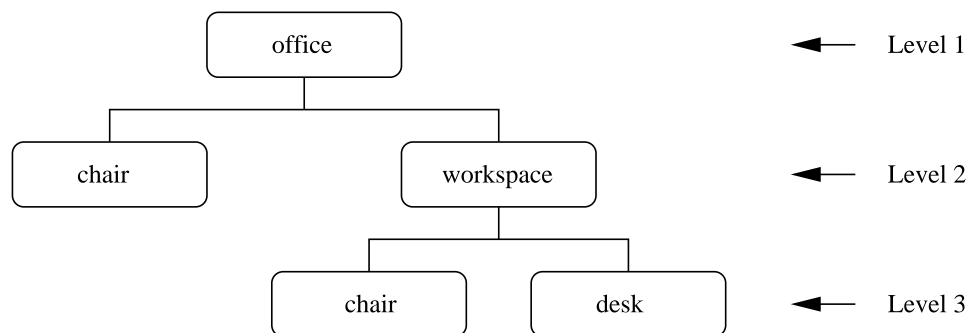


Figure 12.17. Tree hierarchy of placement of spaces and furniture in a simple office.

Here, office and workspace are composite hierarchy objects (i.e., think of them as containers) and chair and table are components that will be positioned as leafs in the component hierarchy tree.

The input/output is as follows:

```
print >> java TestComposite
*** Start level 1 : Office
*** Chair: Customer Chair
*** Start level 2 : Workspace
*** Chair: Office Chair
*** Table: Office Desk
*** Finish level 2 : Workspace
*** Finish level 1 : Office
print >>
```

The composite object `office` is the root of the space hierarchy.

Example 06. Draw Composite Hierarchy of Furniture

In this example, we assemble a composite hierarchy for a dining room that contains four eating areas, and then systematically traverse the composite hierarchy tree to draw the dining room exterior and contents.

Figure 12.18 shows geometry of the dining room exterior and layout of tables and chairs in the dining room.

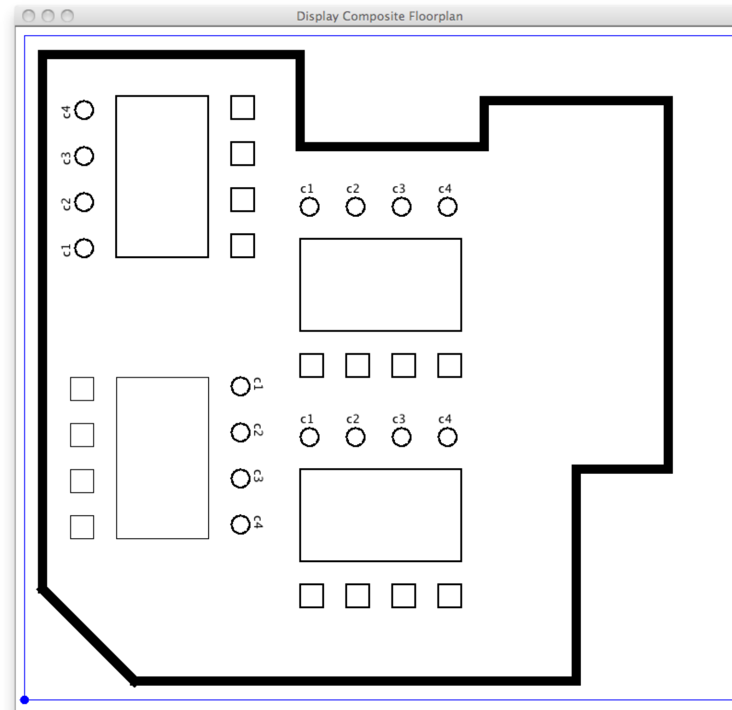


Figure 12.18. Modeling the perimeter of a room and layout of furniture as a composite hierarchy.

There a number of ways to set up this problem. One possibility is to ...

> **... specify the position and orientation of every table and chair object with respect to a single coordinate system.**

While the use of a single coordinate simplifies source code needed to draw the dining room layout, this approach completely ignores the local relationships between objects needed to implement required functionality. For example, the chairs need to be positioned relative to the table so that people can sit at the table – if at some point the table moves, then the chairs need to be moved as well. This suggests that a far better approach is to ...

> **... implement the system layout with a hierarchy of coordinate systems, and then devise code to manage the manipulation of coordinate systems during traversal of the composite hierarchy.**

The source code is arranged into four groups of files, defintion of leaf components (e.g., rectangular boxes), support for a composite hierarchy of graphical interfaces, drawing of the composite hierarchy, and the test program.

```
------------------------------------------------------
Leaf components              Composite Hierarchy
======================================================

Box.java                     CompositeHierarchy.java
Ellipse.java                 Graphic.java
Polygon.java

------------------------------------------------------
Graphical Interface          Test Program
======================================================
DisplayCompositePanel.java   TestComposite.java
DisplayComposite.java

======================================================
```

Figure 12.19 shows the relationship among classes and interfaces in the composite hierarchy.

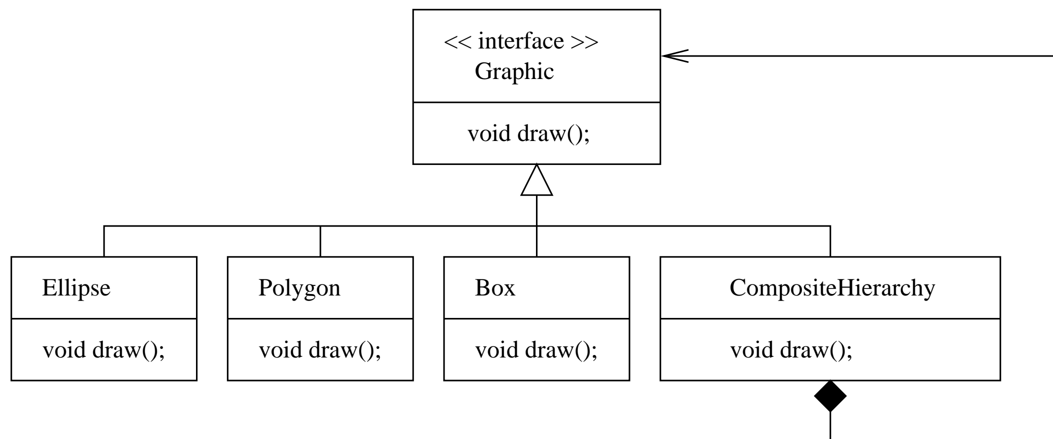


Figure 12.19. Composite hierarchy for modeling the perimeter of a room and layout of furniture.

Drawing of components is facilitated through the definition of a graphic interface,

```
/*
 *  ==========================================================
 *  Graphic.java: Graphic interface ....
 *  ==========================================================
 */

import java.awt.*;
import java.awt.geom.*;

public interface Graphic {
    public void draw( Graphics2D g2 ); // Draw the graphic ...
}
```

followed by an Ellipse class that implements the Graphic interface, i.e.,

```
/*
 *  ==============================================================================
 *  Ellipse.java: Draw ellipse component. This is a leaf in the component hierarchy.
 *  ==============================================================================
 */

import java.awt.*;
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.*;
import java.awt.Point;
import javax.swing.*;

public class Ellipse implements Graphic {
    final static BasicStroke stroke = new BasicStroke(2.0f);
    String sName;
    int x, y;

    public Ellipse ( String sName, int x, int y ) {
       this.sName  = sName;
       this.x = x;
       this.y = y;
    }

    public void draw( Graphics2D g2D ) {

       // Draw ellipse ....

       g2D.setStroke(stroke);
       g2D.draw(new Ellipse2D.Double(x, y, 20, 20));

       // Add label to ellipse ...

       g2D.scale( 1.0, -1.0 );
       g2D.drawString( sName, x, - 200);
       g2D.scale( 1.0, -1.0 );
    }
}
```

The framework for creating a hierarchy of Graphic entities is defined as follows:

```
/*
 *  ================================================================
 *  CompositeHierarchy.java:
 *  ================================================================
 */

import java.lang.Math;
import java.util.List;
import java.util.ArrayList;
```

```
import java.awt.geom.*;   // Needed for affine transformation....
import java.awt.Graphics;
import java.awt.Graphics2D;

public class CompositeHierarchy implements Graphic {
    private static int iCurrentLevel = 0;
    private static double dCurrentXOffSet  = 0.0;
    private static double dCurrentYOffSet  = 0.0;
    private static double dCurrentRotation = 0.0;
    private static AffineTransform at = new AffineTransform();

    // Coordinates and orientation of composite graphic ...

    private double     xOffset;
    private double     yOffset;
    private double    rotation;

    // Collection of child graphics.

    private ArrayList<Graphic> children = new ArrayList<Graphic>();

    // Constructor for composite hierarchy ...

    public CompositeHierarchy( double xOffset, double yOffset, double rotation ) {
       this.xOffset  = xOffset;
       this.yOffset  = yOffset;
       this.rotation = rotation;
    }

    // Set Affine Fransformation  ...

    public void setAffineTransform ( AffineTransform at ) {
       this.at = at;
    }

    // Draw the composite graphic.

    public void draw( Graphics2D g2 ) {

        // Update parameters ...

        iCurrentLevel = iCurrentLevel + 1;
        dCurrentXOffSet  = dCurrentXOffSet  + xOffset;
        dCurrentYOffSet  = dCurrentYOffSet  + yOffset;
        dCurrentRotation = dCurrentRotation + rotation;
        at.translate( xOffset, yOffset );
        at.rotate( rotation );
        g2.setTransform( at );

        // Draw children ....

        for (Graphic graphic : children) {
             graphic.draw( g2 );
        }
```

```
        // Decrement level, x- and y- offsets and rotation....

        iCurrentLevel    = iCurrentLevel - 1;
        dCurrentXOffSet  = dCurrentXOffSet  - xOffset;
        dCurrentYOffSet  = dCurrentYOffSet  - yOffset;
        dCurrentRotation = dCurrentRotation - rotation;
        at.rotate( -rotation );
        at.translate( -xOffset, -yOffset );
        g2.setTransform( at );
    }

    // Adds the graphic to the composition.

    public void add(Graphic graphic) {
        children.add(graphic);
    }

    //Removes the graphic from the composition.

    public void remove(Graphic graphic) {
        children.remove(graphic);
    }
}
```

Display Composite Hierarchy

Visualization of the composite hierarchy is handled by two source code files, DisplayComposite.java nad DisplayCompositePanel.java, which, in turn, are called by the test program.

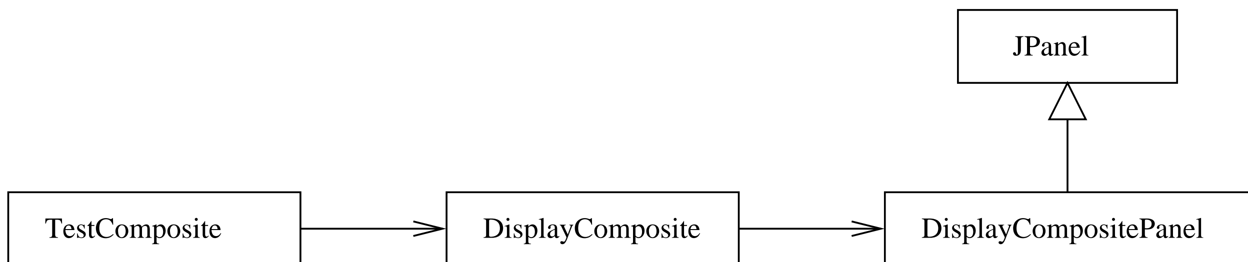


Figure 12.20. Relationship among classes in display of the floor plan.

The relationship among these files, and the point of extension to create a panel for display of the floor plan is shown in Figure 12.20.

```
/*
 *  ========================================================================
 *  DisplayComposite.java: .....
 *  ========================================================================
 */

import java.awt.*;
import java.awt.Color;
import javax.swing.*;
```

```
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.*;
import java.awt.Point;
import java.awt.event.*;
import java.util.*;

public class DisplayComposite {
   private DisplayCompositePanel canvas;
   private CompositeHierarchy      base;

   public DisplayComposite( CompositeHierarchy base ) {
      this.base = base;
   }

   // Create graphics screen ....

   public void display() {
       canvas = new DisplayCompositePanel( base );
       canvas.setBackground( Color.white );

       // Create a scroll pane and add the panel to it.

       JScrollPane scrollCanvas = new JScrollPane( canvas,
                   JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
                   JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
       scrollCanvas.setPreferredSize( new java.awt.Dimension(800, 800) );

       // Create buttons for panel along bottom of screen ...

       JPanel panel = new JPanel();
       panel.setLayout( new BorderLayout() );
       panel.add( scrollCanvas, BorderLayout.CENTER );

       JFrame frame = new JFrame("Display Composite Floorplan");
       frame.getContentPane().setLayout( new BorderLayout() );
       frame.getContentPane().add( panel );
       frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
       frame.setSize( 800, 800);
       frame.pack();
       frame.setVisible(true);
   }

   public void border() {
       canvas.drawBorder();
   }
}
```

and

```
/*
 *  =====================================================================
 *  DisplayCompositePanel.java: A simple JPanel to display contents of the
 *                              composite hierarchy.
```

```
 *  ======================================================================
 */

import java.awt.*;
import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.geom.*;
import java.awt.Point;
import java.awt.event.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.beans.PropertyChangeEvent;
import java.util.*;
import java.util.List;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Iterator;
import java.util.Set;
import javax.swing.*;

public class DisplayCompositePanel extends JPanel {
  private CompositeHierarchy base;
  private AffineTransform      at;
  int iBorder = 10;
  int width, height;
  Graphics2D g2D;

  public DisplayCompositePanel( CompositeHierarchy base ) {
     this.base = base;
  }

  // Paint panel ...

  public void paint() {
     Graphics g = getGraphics();
     super.paintComponent(g);
     paintComponent(g);
  }

  protected void paintComponent(java.awt.Graphics g) {
     Dimension d = getSize();
     height = getSize().height;
     width  = getSize().width;
     g2D = (Graphics2D) g.create();

     // Create screen-to-base coordinate transformation ...

     at = new AffineTransform();
     at.translate( iBorder, height - iBorder );
     at.scale( 1, -1 );
     g2D.setTransform( at );
```

```
      // Draw border around bounding box ...

      drawBorder();

      // Draw contents of composite hierarchy ...

      base.setAffineTransform( at );
      base.draw( g2D );
   }

   // Draw rectangle around exterior boundary ...

   protected void drawBorder() {
      int x1 = 0;
      int y1 = 0;
      int x2 = width - 2*iBorder;
      int y2 = height- 2*iBorder;

      g2D.setColor( Color.blue );
      g2D.draw( new Line2D.Double( x1, y1, x2, y1 ));
      g2D.draw( new Line2D.Double( x1, y1, x1, y2 ));
      g2D.draw( new Line2D.Double( x1, y2, x2, y2 ));
      g2D.draw( new Line2D.Double( x2, y1, x2, y2 ));
      g2D.setColor( Color.blue );
      g2D.fill(new Ellipse2D.Double( -5,  -5, 10, 10));
   }
}
```

Test Program. ....

```
/*
 * ======================================================================
 * TestComposite.java: Test program for drawing the exterior boundary and
 *                     layout of furniture in a dining room.
 * ======================================================================
 */

import java.util.ArrayList;

public class TestComposite {
   public static void main(String[] args) {

   // Initialize three composite graphics

   CompositeHierarchy base = new CompositeHierarchy( 0.0, 0.0, 0.0 );

   CompositeHierarchy eatingArea1 = new CompositeHierarchy(  50.0, 350.0, -Math.PI/2.0 );
   CompositeHierarchy eatingArea2 = new CompositeHierarchy( 300.0, 100.0,          0.0 );
   CompositeHierarchy eatingArea3 = new CompositeHierarchy( 250.0, 480.0,  Math.PI/2.0 );
   CompositeHierarchy eatingArea4 = new CompositeHierarchy( 300.0, 350.0,          0.0 );

   // Create polygon for room exterior ...

   ArrayList<Integer> x = new ArrayList();
```

```
   ArrayList<Integer> y = new ArrayList();

   x.add (  20 ); y.add ( 120 ); x.add (  20 ); y.add ( 700 );
   x.add ( 300 ); y.add ( 700 ); x.add ( 300 ); y.add ( 600 );
   x.add ( 500 ); y.add ( 600 ); x.add ( 500 ); y.add ( 650 );
   x.add ( 700 ); y.add ( 650 ); x.add ( 700 ); y.add ( 250 );
   x.add ( 600 ); y.add ( 250 ); x.add ( 600 ); y.add (  20 );
   x.add ( 120 ); y.add (  20 );

   Polygon room = new Polygon(   0, 0, x, y );

   // Create table and chairs ...

   Box table  = new Box(   0,  50, 175, 100);
   Box chair1 = new Box(   0,   0,  25,  25);
   Box chair2 = new Box(  50,   0,  25,  25);
   Box chair3 = new Box( 100,   0,  25,  25);
   Box chair4 = new Box( 150,   0,  25,  25);
   Ellipse chair5 = new Ellipse("c1",   0, 175);
   Ellipse chair6 = new Ellipse("c2",  50, 175);
   Ellipse chair7 = new Ellipse("c3", 100, 175);
   Ellipse chair8 = new Ellipse("c4", 150, 175);

   // Define layout of table and chairs ....

   CompositeHierarchy tableandchairs = new CompositeHierarchy( 0.0, 0.0, 0.0 );
   tableandchairs.add (table);
   tableandchairs.add (chair1);  tableandchairs.add (chair2);
   tableandchairs.add (chair3);  tableandchairs.add (chair4);
   tableandchairs.add (chair5);  tableandchairs.add (chair6);
   tableandchairs.add (chair7);  tableandchairs.add (chair8);

   // Add table and chairs to eating areas ....

   eatingArea1.add( tableandchairs );
   eatingArea2.add( tableandchairs );
   eatingArea3.add( tableandchairs );
   eatingArea4.add( tableandchairs );

   // Compose the scene ....

   base.add(eatingArea1);
   base.add(eatingArea2);
   base.add(eatingArea3);
   base.add(eatingArea4);
   base.add(room);

   // Draw complete graphic starting at the base ...

   DisplayComposite display = new DisplayComposite( base );
   display.display();
   }
}
```

Points to note:

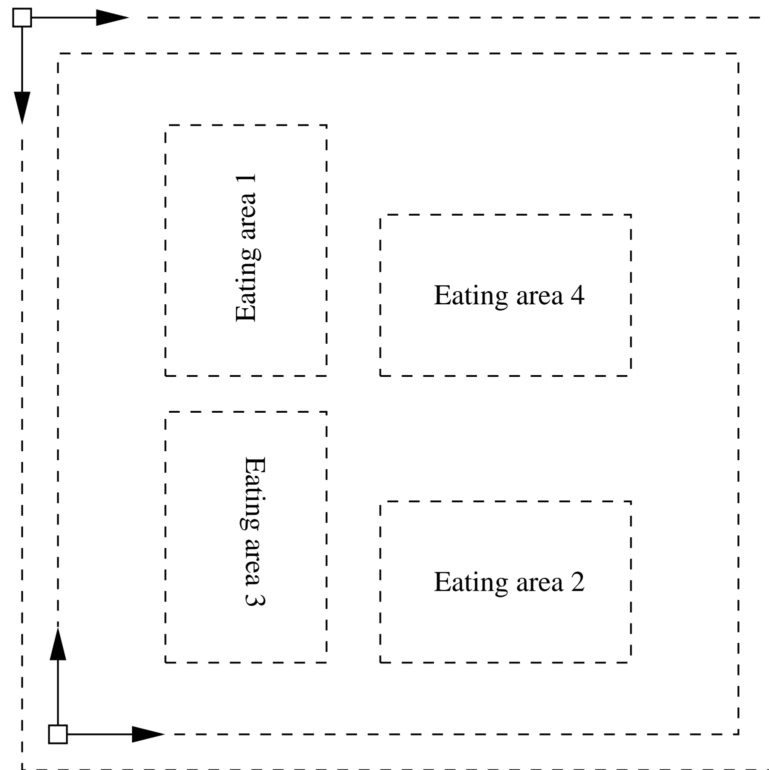


Figure 12.21. Coordinate system hierarchy for display of furniture placement in a dining room.

1. We can add print statements to the composite hierarchy to track the systematic evolution of coordinate offsets and rotations that occur during traversal of the composite hierarchy. This gives:

```
Script started on Thu Feb 23 10:09:43 2012
prompt >> java TestComposite
Start : Level=  1
      : (x,y,rot) offset = (   0.0,   0.0,  0.00)
        AffineTransform[ [ 1.0,  0.0,  10.0 ],
                         [ 0.0, -1.0, 786.0 ]]
Start : Level=  2
      : (x,y,rot) offset = (  50.0, 350.0, -1.57)
        AffineTransform[ [-0.0,  1.0,  60.0 ],
                         [ 1.0,  0.0, 436.0 ]]
Start : Level=  3
      : (x,y,rot) offset = (  50.0, 350.0, -1.57)
        AffineTransform[ [-0.0,  1.0,  60.0 ],
                         [ 1.0,  0.0, 436.0 ]]
End Level:  3
End Level:  2

Start : Level=  2
      : (x,y,rot) offset = ( 300.0, 100.0,  0.00)
        AffineTransform[ [ 1.0,  0.0, 310.0 ],
                         [ 0.0, -1.0, 686.0 ]]
```

```
Start : Level=  3
      : (x,y,rot) offset = ( 300.0, 100.0,  0.00)
        AffineTransform[ [ 1.0,  0.0, 310.0 ],
                         [ 0.0, -1.0, 686.0 ]]
End Level:  3
End Level:  2

Start : Level=  2
      : (x,y,rot) offset = ( 250.0, 480.0,  1.57)
        AffineTransform[ [ 0.0, -1.0, 260.0 ],
                         [-1.0, -0.0, 306.0 ]]
Start : Level=  3
      : (x,y,rot) offset = ( 250.0, 480.0,  1.57)
        AffineTransform[ [ 0.0, -1.0, 260.0 ],
                         [-1.0, -0.0, 306.0 ]]
End Level:  3
End Level:  2

Start : Level=  2
      : (x,y,rot) offset = ( 300.0, 350.0,  0.00)
        AffineTransform[ [ 1.0,  0.0, 310.0 ],
                         [ 0.0, -1.0, 436.0 ]]
Start : Level=  3
      : (x,y,rot) offset = ( 300.0, 350.0,  0.00)
        AffineTransform[ [ 1.0,  0.0, 310.0 ],
                         [ 0.0, -1.0, 436.0 ]]
End Level:  3
End Level:  2
End Level:  1
prompt >>
Script done on Thu Feb 23 10:09:58 2012
```

2.

3.

4.

12.6 Data-Flow Processing with Graphs of Executable Components

In this section we present interface specifications and code implementations for a general purpose, executable, data processing component. The component is assembled from wires and ports. Data processing is driven by a component engine infrastructure.

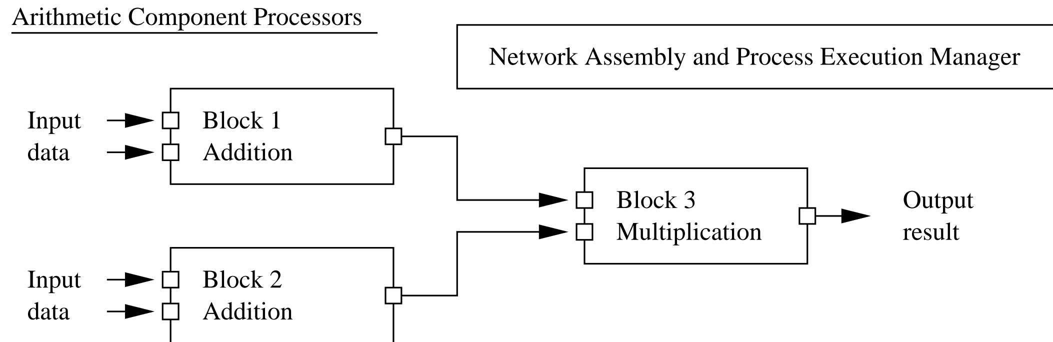


Figure 12.22. Component network of arithmetic processors.

For example, if BC is a basic component, ...

```
[java] BC(Block 1: ADD)  Inputs: 1.0 2.0  Outputs: 3.0
[java] BC(Block 2: ADD)  Inputs: 3.0 4.0  Outputs: 7.0
[java] BC(Block 3: MULT)  Inputs: 3.0 7.0  Outputs: 21.0
[java] *** i =  1 Answer: y = 21.00
```

When the network description is enclosed within a looping construct, repeated evaluations of the arithmetic network are possible, e.g.,

```
Loop 1
[java] BC(Block 1: ADD)  Inputs: 2.0 3.0  Outputs: 5.0
[java] BC(Block 2: ADD)  Inputs: 4.0 5.0  Outputs: 9.0
[java] BC(Block 3: MULT)  Inputs: 5.0 9.0  Outputs: 45.0
[java] Answer: y = 45.00

Loop 2
[java] BC(Block 1: ADD)  Inputs: 3.0 4.0  Outputs: 7.0
[java] BC(Block 2: ADD)  Inputs: 5.0 6.0  Outputs: 11.0
[java] BC(Block 3: MULT)  Inputs: 7.0 11.0  Outputs: 77.0
[java] Answer: y = 77.00
```

The network process manager is responsible for incrementally stepping the data processing components through computational steps. For example, in Figure 12.22, one step of computation is required to compute the sums in blocks 1 and 2. A second computational step computes the multiplication in block 3, leading to the output result.

Source Code

From a software standpoint, this application is interesting because ...

... it is completely defined by collections of interfaces for the components, ports, wires and processing elements.

The source code is an assembly of 13 files. To simplify the explanation of the interface definitions and implementation, we will the bundle source code into interfaces and implementations for ports, wires, and components.

```
Part1: Port Implementation          Port Interfaces
========================================================

       PortImpl.java                Port.java
       InputPortImpl.java           InputPort.java
       OutputPortImpl.java          OutputPort.java

Part2: Component Implementations    Component Interfaces
========================================================

       BaseComponent.java           Component.java
                                    ComponentEngine.java

Part3: Wire Implementation          Wire Interface
========================================================

       WireImpl.java                Wire.java

Part4: Management of Data Processing
========================================================
       MetaComponentSimple.java

========================================================
```

We will see that the system architecture and processing/function capability can be completely defined by ...

... relationships among the port, component, and wire interfaces.

which, in turn, need to reflected in ...

... implementations for the component, port, wire and computational engine classes.

Network of Arithmetic Processors

To demonstrate these capabilities, a network of arithmetic block processors is defined by the three files:

```
Part5: Application Implementation         Use of Interfaces
==============================================================
       ArithmeticComponentEngine.java     ComponentEngine.java
       ArithmenticOps.java
       TestNetwork.java
==============================================================
```

ArithmeticOps.java is a simple implementation for basic arithmetic operations. ArithmeticComponentEngine.java is the processing engine that retrieves data values from the incoming ports and calls the arithmetic operations processor. TestNetwork.java is the code that assembles the component network of arithmetic processors shown in Figure 12.22.

Part 1. Port Interface Hierarchy and Implementation

Port.java, InputPort.java and OutputPort.java. Input and output ports are defined through a two-level hierarchy of interface definitions.

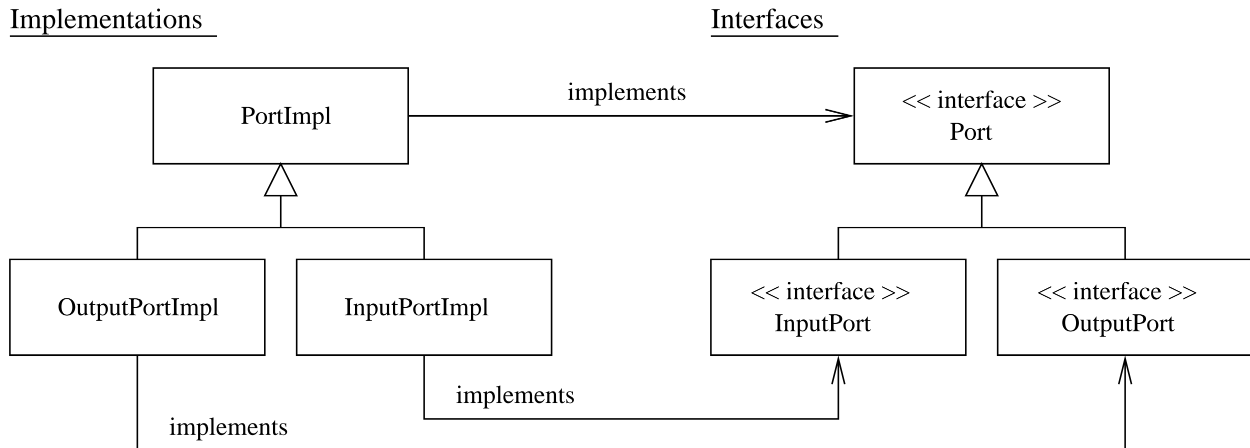


Figure 12.23. Class and interface architecture for ports.

The bundled source code is as follows:

source code

```
/**
 * ================================================================
 * Port.java, InputPort.java and OutputPort.java ...
 * ================================================================
 */

package component;

public interface Port<T> {
   public Component<T> getParent();
   public void setName(String sName);
}

// Set the name and value of the input port ...

public interface InputPort<T> extends Port<T> {
   public void setValue(T value);
}

// Get the value of the output port
```

```
public interface OutputPort<T> extends Port<T> {
   public T getValue();
}
```

Points to note are as follows:

1. The port interface specifies a method for retrieving the component to which a port belongs (i.e., getParent()) and a simple method for setting the name of the port. Notice that the declaration doesn't talk about a specific component implementations – instead it ...

   ... simply refers to the parent component indirectly through the Component interface implemented by specific components.

2. The input and output port interfaces extend (or build upon) the port interfaces. Input ports set data values and output ports get data values.

3. The use of java generics implies that all three port interfaces will be implemented with a single data type T (e.g., Boolean is true/false values are propogated through the network; Double if numerical values are propogated through the network).

Port Implementation.

source code

```
/**
 *  ===============================================================
 *  PortImpl.java: A base for building port implementations. It can
 *  function either as an input or output port, or both.
 *
 *    @see com.wickedcooljava.sci.component.InputPortImpl
 *    @see com.wickedcooljava.sci.component.OutputPortImpl
 *  ===============================================================
 */

package component;

public class PortImpl<T> implements Port<T> {
   private boolean initialized = false;
   private String        sName;
   private Component<T> parent;
   private T value;

   public PortImpl(Component<T> par) {
      parent = par;
   }

   public Component<T> getParent() {
      return parent;
   }
```

```
   public void setName( String sName ) {
      this.sName = sName;
   }

   public void setValue(T val) {
      this.initialized = true;
      value = val;
   }

   public T getValue() {
      return value;
   }

   public boolean getInitialized() {
      return this.initialized;
   }

   public String toString() {
      return "\n" + sName + ".value=" + value;
   }
}
```

The methods getParent() and setName() are provided in PortImpl to satisfy the contract specified in Port.java. Notice that we also provide methods for setValue() and getValue() – these could have been provided in InputPort and OutputPort. The method getInitialized() is used by component engines to ensure that data values are initialized before proceeding with a computation.

Input and Output Port Implementations.

Implementations of InputPort and OutputPort use PortImpl as a base. Thus, the bundled source code is as follows:

source code

```
package component;

public class InputPortImpl<T> extends PortImpl<T> implements InputPort<T> {
   public InputPortImpl( Component<T> parent ) {
      super(parent);
   }
}

public class OutputPortImpl<T> extends PortImpl<T> implements OutputPort<T> {
   public OutputPortImpl( Component<T> parent ) {
      super(parent);
   }
}
```

Since all of the functionality for setting and getting data values has been provided in PortImpl.java, all ImportPortImpl and OutputPortImpl have to do is provide constructor methods that pass a reference to the parent component to the constructor method in PortImpl.java.

Part 2. Component Interface and Base Component Implementation

Component.java. A component has an specific number of input and output ports that connect to other components, and performs some process that converts inputs into outputs.

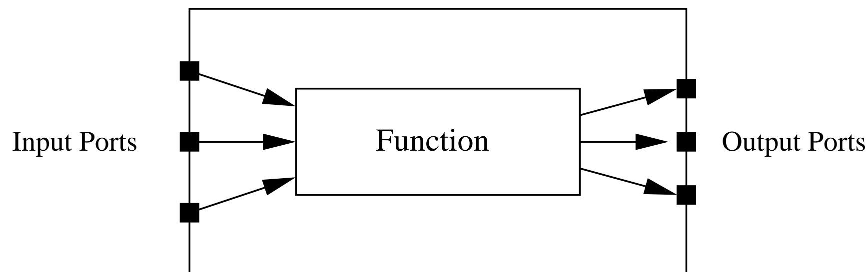


Figure 12.24. Architecture of a data processing component.

Here is the component interface definition:

source code

```
/**
 * ===================================================================================
 * Component.java: A generic interface for components that process any type of data.
 * ===================================================================================
 */

package component;

public interface Component<T> {
   // set component name ...
   public void setName( String sName );

   // get the number of input ports
   public int getInputSize();

   // get the number of output ports
   public int getOutputSize();

   // get the nth input port
   public InputPort<T>  getInputPort(int index);

   // get the nth output port
   public OutputPort<T> getOutputPort(int index);

   // perform the component's processing
   public void process();
}
```

Base Component Implementation

The base component implementation provides a pass-through implementation of the Component interface, i.e.,

source code

```
/**
 *  =====================================================================
 *  BaseComponent.java: Basic implementation of the Component interface.
 *  =====================================================================
 */

package component;

public class BaseComponent<T> implements Component<T> {
   protected String sName;

   protected int inSize, outSize;
   protected InputPortImpl<T>[]   inports;
   protected OutputPortImpl<T>[] outports;

   // the function performed by this component

   protected ComponentEngine<T> function;

   /**
     *
     * @param inputs Number of input ports
     * @param outputs Number of output ports
     * @param f The function to perform the processing
     */

   public BaseComponent(int inputs, int outputs, ComponentEngine<T> f ) {
      inSize   = inputs;
      outSize  = outputs;
      function = f;

      inports = new InputPortImpl[inSize];
      for (int i = 0; i < inSize; i++) {
         inports[i] = new InputPortImpl<T>(this);
      }

      outports = new OutputPortImpl[outSize];
      for (int i = 0; i < outSize; i++) {
         outports[i] = new OutputPortImpl<T>(this);
      }
   }

   public void setName ( String sName ) {
           this.sName = sName;
           for (int i = 0; i < inSize; i++) {
               inports[i].setName("BC(" + sName + ").in(" + i + "]");
           }
```

```
            for (int i = 0; i < outSize; i++) {
                outports[i].setName("BC(" + sName + ").out[" + i + "]");
            }
    }

    public int getInputSize() {
       return inSize;
    }

    public int getOutputSize() {
       return outSize;
    }

    public InputPort<T> getInputPort(int index) {
       return inports[index];
    }

    public OutputPort<T> getOutputPort(int index) {
       return outports[index];
    }

    /**
      * Delegate to the engine to do the processing.
      */

    public void process() {
       function.process(inports, outports);
    }

    // Detailed string representation ....

    public String toString() {
       StringBuffer buf = new StringBuffer();
       buf.append("BC(" + sName + ") ");
       buf.append(" Inputs: ");

       for (PortImpl port : inports) {
          buf.append(port.getValue());
          buf.append(" ");
       }
       buf.append(" Outputs: ");

       for (OutputPort port : outports) {
          buf.append(port.getValue());
          buf.append(" ");
       }
       return buf.toString();
    }
}
```

plus provision for evaluation of arithmetic functions through the use of ComponentEngine interfaces, i.e.,

```
   protected ComponentEngine<T> function;
```

Details on the component engine interface are provided in the next section.

Part 3. Wire Interface and Implementation A wire connects an output port to one or more input ports.

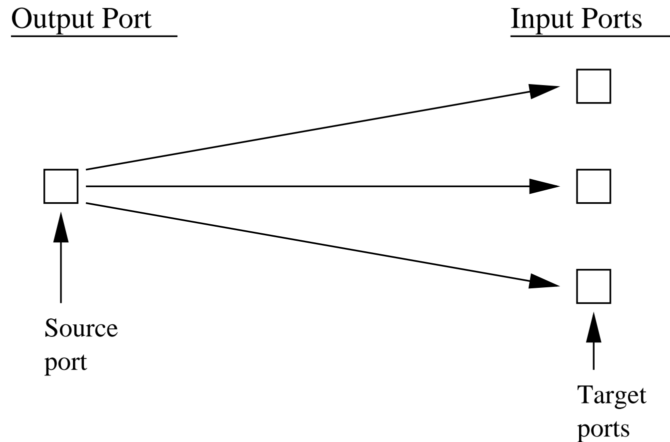


Figure 12.25. Wire model: one-to-many connectivity between a source port and target ports.

Here is a schematic of the implementation for the wire interface:

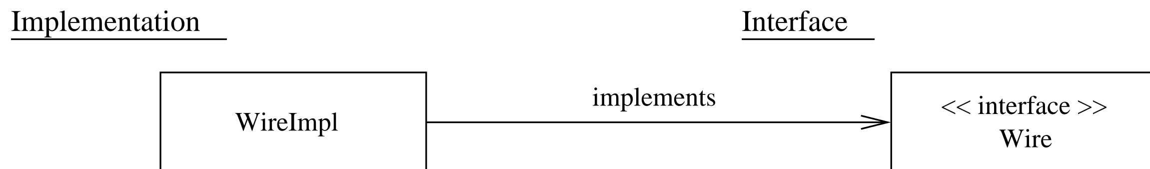


Figure 12.26. Implementation of the wire interface.

Wire.java. The wire interface contains methods for retrieving the source port, the number of target ports, a specific target port, and a method for propogating data from the source port to one or more target ports.

source code

```
/**
 *  ==========================================================
 *  Wire.java: Interface for a wire ...
 *  ==========================================================
 */

package component;

public interface Wire<T> {
   public OutputPort<T> getSourcePort();
```

```
   public int  getNumberOfTargetPorts();
   public InputPort<T>  getTargetPort(int index);
   public void propagateSignal();
}
```

Wire Implementation

The wire implementation uses arraylists to support the one-to-many relationship between source and destination ports.

source code

```
/**
 *  ==========================================================================
 *  WireImpl.java: A basic implementation of a Wire.
 *
 *  Note. The number of target ports can grow dynamically, but for performance
 *  reasons it will use a single target object when the first one is added.
 *  As more targets are added, an ArrayList is used and is populated with them.
 *  ==========================================================================
 */

package component;

import java.util.ArrayList;

public class WireImpl<T> implements Wire<T> {
   private OutputPort<T> source;

   // lazy target array

   private ArrayList<InputPort<T>> targetList;
   private InputPort<T> target;
   private int count = 0;

   public WireImpl(OutputPort<T> src) {
      source = src;
   }

   public OutputPort<T> getSourcePort() {
      return source;
   }

   public int getNumberOfTargetPorts() {
      return count;
   }

   public InputPort<T> getTargetPort(int index) {
      if (index >= count || index < 0) {
          throw new IndexOutOfBoundsException();
      }

      if (target != null) {
```

```
            return target;
        }

        return targetList.get(index);
    }

    public void addTargetPort(InputPort<T> tgt) {
        if (targetList == null) {
            if (target == null) {
                target = tgt;
                count++;
            } else {
                targetList = new ArrayList<InputPort<T>>();
                targetList.add(target);
                target = null;
            }
        }

        if (targetList != null) {
            if (!targetList.contains(tgt)) {
                targetList.add(tgt);
                count++;
            }
        }
    }

    public void propagateSignal() {
        T value = source.getValue();

        if (target == null) {
            if (targetList != null) {
                for (InputPort<T> tgt : targetList) {
                     tgt.setValue(value);
                }
            }
        } else {
            target.setValue(value);
        }
    }
}
```

Part 4. Simple Implementation of a MetaComponent

In this section we provide a JGraphT-enabled implementation for assembly of components into a network of processes, and systematic evaluation (process execution) of arthmetic expressions.

The implementation is “simple” in the sense that MetaComponentSimple does not implement the Component interface – as such, this implementation cannot be inserted into a hierarchy of arithmetic processors.

source code

```
/**
```

```
 * ===================================================================
 * MetaComponentSimple.java: This is a simple version of a MetaComponent.
 *
 * It uses jgrapht to maintain the connections between child components.
 *
 * Original Code: Wicked Cool Java book.
 * Modified by: Mark Austin                              October 2009
 * ===================================================================
 */

package component;

import org.jgraph.*;
import org.jgraph.graph.*;

import org.jgrapht.*;
import org.jgrapht.ext.*;
import org.jgrapht.graph.*;

import org.jgrapht.ListenableGraph;
import org.jgrapht.ext.JGraphModelAdapter;
import org.jgrapht.graph.ListenableDirectedGraph;

// Setup default edge

import org.jgrapht.graph.DefaultEdge;

public class MetaComponentSimple<T> {

    // the graph that maintains child components

    private ListenableDirectedGraph graph;

    public MetaComponentSimple() {
        graph = new ListenableDirectedGraph<T,DefaultEdge>(DefaultEdge.class);
    }

    // ================================================================
    // Connect an output port to an input port.
    // ================================================================

    public void connect( OutputPort<T> out, InputPort<T> in ) {

        Component<T> source = out.getParent();
        Component<T> target =  in.getParent();

        // 1: Add parent components to graph

        if (graph.containsVertex(source) != true) {
            graph.addVertex(source);
        }

        if (graph.containsVertex(target) != true) {
            graph.addVertex(target);
        }
```

```
      // 2: Add ports to graph

      if (graph.containsVertex(in) != true ) {
          graph.addVertex(in);
      }

      if (graph.containsVertex(out) != true) {
          graph.addVertex(out);
      }

      // 3: Add an edge from out parent to output port

      graph.addEdge(source, out);

      // 4: Add an edge from output port to input port

      graph.addEdge(out, in);

      // 5: add an edge from input port to target component

      graph.addEdge(in, target);
   }

   // ======================================================================
   // Perform the processing by processing each of the subcomponents
   // and propagating signals from outputs to inputs.
   // ======================================================================

   public void process() {
      processSubComponents();
      propagateSignals();
   }

   // ======================================================================
   // For all connected sub componets, propagate signals from all outputs
   // to all inputs.
   // ======================================================================

   private void propagateSignals() {

      // Walk along edges and propogate output port values to
      // input port values ....

      for (Object item : graph.edgeSet()) {

         DefaultEdge edge = (DefaultEdge) item;
         Object source = graph.getEdgeSource( edge );
         Object target = graph.getEdgeTarget( edge );

         if (source instanceof OutputPort) {
            OutputPort<T> out = (OutputPort<T>) source;
            InputPort<T>   in = (InputPort<T>) target;
            in.setValue( out.getValue() );
         }
```

```
      }
   }

   // ==================================================================
   // Process all subcomponents, by calling the process methods for each.
   // ==================================================================

   private void processSubComponents() {
      for (Object item : graph.vertexSet()) {
          if (item instanceof Component) {
              ((Component<T>) item).process();
          }
      }
   }

   // ==================================================================
   // Returns the graph used by this MetaComponent
   // ==================================================================

   public Graph getGraph() {
      return graph;
   }
}
```

A few points to note:

1. The connect() method is responsible for assembly of the JGraphT model. Vertices are added to the graph for the parent components (i.e., in.getParent()) and out.getParent()) and the input/output ports connected to wires. Input ports that are not connected to a wire are not part of the graph model.

2. The process() method systematically executes functionality for all of the subcomponents (i.e., component fuctionalities in the graph), and then propagates data values across the wires that are connected to ports.

Example 07. Network of Arithmetic Block Processors

ComponentEngine.java. This class defines an interface for the processing engine of a component. The purpose of the component engine is to convert inputs into outputs.

source code

```
package component;

public interface ComponentEngine<T> {
   public void process( PortImpl<T>[] in, PortImpl<T>[] out );
}
```

ArithmeticOps.java. A simple implementation for arithmetic operations.

source code

```
/**
 *  ==================================================================
 *  ArithmeticOps.java: A simple arithmetic operations implementation
 *  ==================================================================
 */

package testnetwork;

public class ArithmeticOps {
   private String name;
   private ArithmeticOps ops;

   // Details of arithmetic operation ....

   public static final ArithmeticOps      ADD = new ArithmeticOps("ADD");
   public static final ArithmeticOps MULTIPLY = new ArithmeticOps("MULTIPLY");
   public static final ArithmeticOps SUBTRACT = new ArithmeticOps("SUBTRACT");
   public static final ArithmeticOps   DIVIDE = new ArithmeticOps("DIVIDE");

   // =====================================================================
   // Create arithmetic operations object with the specified number of args..
   // =====================================================================

   public ArithmeticOps(int inputs, int outputs) {
      this.name = null;
   }

   public ArithmeticOps( String name ) {
      this.name = name;
   }

   public void setOpsType ( final ArithmeticOps ops ) {
      this.ops = ops;
   }

   public void setName ( String name ) {
      this.name = name;
   }

   public String        getName()  { return this.name; }
   public ArithmeticOps getOps()   { return this.ops; }

   public double computeValue ( double arg1, double arg2 ) {
      double value = 0.0;

      if ( ArithmeticOps.ADD == this.getOps() ) {
         value = arg1 + arg2;
      } else if ( ArithmeticOps.SUBTRACT == this.getOps() ) {
         value = arg1 - arg2;
      } else if ( ArithmeticOps.MULTIPLY == this.getOps() ) {
         value = arg1 * arg2;
      } else if ( ArithmeticOps.DIVIDE == this.getOps() ) {
```

```
         value = arg1 / arg2;
      } else {
         System.out.println("*** Arithmetic Operation Type not defined .. ");
      }

      return value;
   }

   public String toString() {
      String s = "ArithmeticOps: type = " + this.name + "\n";
      return s;
   }
}
```

ArithmeticOpsEngine.java. This class defines a component engine for arithmetic operations. It implements the ComponentEngine interface.

source code

```
/**
 *  ============================================================================
 *  ArithmeticComponentEngine.java: A component engine for computing arithmetic
 *                                  operations.
 *  ============================================================================
 */

package testnetwork;

import component.*;

public class ArithmeticComponentEngine implements ComponentEngine<Double> {
   private ArithmeticOps tt;

   public ArithmeticComponentEngine( ArithmeticOps aops ) {
      tt = aops;
   }

   public void process( PortImpl<Double>[] in, PortImpl<Double>[] out ) {
      PortImpl<Double> arg1 = in[0];
      PortImpl<Double> arg2 = in[1];

      if( arg1.getInitialized() == true && arg2.getInitialized() == true ) {
          double value = tt.computeValue ( arg1.getValue().doubleValue(),
                                           arg2.getValue().doubleValue() );
          PortImpl<Double> arg3 = out[0];
          arg3.setValue ( value );
      }
   }
}
```

TestNetwork.java. Here is the code to assemble the network of computational blocks, initialize the port data, and propogate the computations through the network of computational blocks.

source code

```
/*
 *  ==========================================================================
 *  TestNetwork.java: Exercise simple metacomponent model in a network of
 *  arithmetic operations...
 *
 *  Modified by: Mark Austin                                      October 2009
 *  ==========================================================================
 */

package testnetwork;

import component.*;

public class TestNetwork {

   // ==============================================
   // Create an Add (ADD) component  ....
   // ==============================================

   public static Component<Double> createAddComponent(int inputs) {

      ArithmeticOps add = new ArithmeticOps (inputs, 1);
      add.setOpsType ( ArithmeticOps.ADD );
      add.setName ( "ADD" );

      ArithmeticComponentEngine processor = new ArithmeticComponentEngine(add);

      BaseComponent<Double> bc = new BaseComponent<Double>(inputs, 1, processor);

      return ( bc );
   }

   // ==============================================
   // Create a Multiply (MULTIPLY) component  ....
   // ==============================================

   public static Component<Double> createMultiplyComponent(int inputs) {

      ArithmeticOps multiply = new ArithmeticOps (inputs, 1);
      multiply.setOpsType ( ArithmeticOps.MULTIPLY );
      multiply.setName ( "MULT" );

      ArithmeticComponentEngine processor = new ArithmeticComponentEngine(multiply);

      BaseComponent<Double> bc = new BaseComponent<Double>(inputs, 1, processor);

      return ( bc );
   }

   // ==============================================
   // Assemble and Exercise component assembly  ....
```

```
// ==============================================

public static void main(String[] args) {

   // Create network manager ....

   MetaComponentSimple<Double> manager = new MetaComponentSimple<Double>();

   // Create Add and Multiply processing components ...

   Component<Double> add1 = createAddComponent(2);
   add1.setName("Block 1: ADD");
   Component<Double> add2 = createAddComponent(2);
   add2.setName("Block 2: ADD");

   Component<Double> mult1 = createMultiplyComponent(2);
   mult1.setName("Block 3: MULT");

   // Get input ports .....

   InputPort<Double> a = add1.getInputPort(0);
   InputPort<Double> b = add1.getInputPort(1);
   InputPort<Double> c = add2.getInputPort(0);
   InputPort<Double> d = add2.getInputPort(1);

   // Set the input values

   a.setValue( new Double(1.0) );
   b.setValue( new Double(2.0) );
   c.setValue( new Double(3.0) );
   d.setValue( new Double(4.0) );

   // Assemble graph of processing components ...

   manager.connect( add1.getOutputPort(0), mult1.getInputPort(0) );
   manager.connect( add2.getOutputPort(0), mult1.getInputPort(1) );

   // Print details of components at beginning of propogation ...

   System.out.println( "");
   System.out.println( "Part 1: Initial Condition of Base Components ... ");
   System.out.println( "------------------------------------------------- ");

   System.out.println(  add1.toString() );
   System.out.println(  add2.toString() );
   System.out.println( mult1.toString() );

   // Print details of meta-component graph ...

   System.out.println( "");
   System.out.println( "Part 2: Meta-Component Graph ... ");
   System.out.println( "---------------------------------- ");

   System.out.println( manager.getGraph().toString() );
```

```
      // Propogate signals through component wires ...

      System.out.println( "");
      System.out.println( "Part 3: Propogate signals: Steps 1 and 2... ");
      System.out.println( "------------------------------------------ ");

      manager.process();
      manager.process();

      // Print details of meta-component graph ...

      System.out.println( "");
      System.out.println( "Part 4: Meta-Component Graph ... ");
      System.out.println( "---------------------------------- ");

      System.out.println( manager.getGraph().toString() );

      System.out.println(  add1.toString() );
      System.out.println(  add2.toString() );
      System.out.println( mult1.toString() );

      System.out.println( "");
      System.out.println( "Part 5: Result .... ");
      System.out.println( "---------------------------------- ");

      OutputPort<Double> y = mult1.getOutputPort(0);
      System.out.println( "Answer: y = " + y.getValue().doubleValue() );
    }
}
```

The (slightly edited) program input/output is as follows:

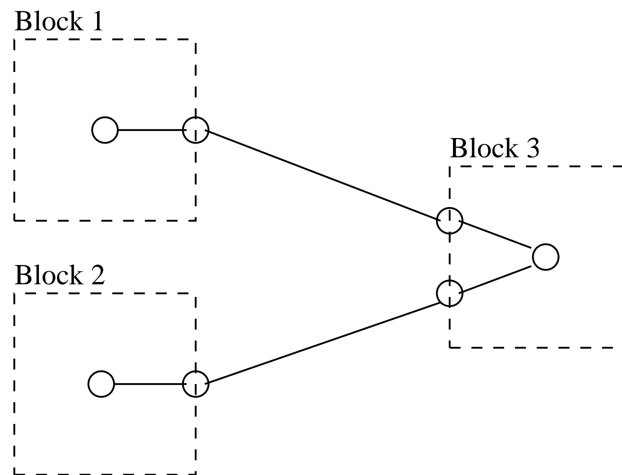
```
prompt >> ant run09
Part 1: Initial Condition of Base Components ...
------------------------------------------------
BC(Block 1: ADD)  Inputs: 1.0 2.0  Outputs: null
BC(Block 2: ADD)  Inputs: 3.0 4.0  Outputs: null
BC(Block 3: MULT)  Inputs: null null  Outputs: null

Part 2: Meta-Component Graph ...
-----------------------------------
( [
  BC(Block 1: ADD)  Inputs: 1.0 2.0  Outputs: null,
  BC(Block 3: MULT) Inputs: null null  Outputs: null ,
  BC(Block 3: MULT).in(0].value=null,
  BC(Block 1: ADD).out[0].value=null,
  BC(Block 2: ADD)  Inputs: 3.0 4.0  Outputs: null ,
  BC(Block 3: MULT).in(1].value=null,
  BC(Block 2: ADD).out[0].value=null
  ] , [
  ( BC(Block 1: ADD) Inputs: 1.0 2.0  Outputs: null,
    BC(Block 1: ADD).out[0].value=null ),
  ( BC(Block 1: ADD).out[0].value=null,
```

```
    BC(Block 3: MULT).in(0].value=null ),
  ( BC(Block 3: MULT).in(0].value=null,
    BC(Block 3: MULT) Inputs: null null Outputs: null ),
  ( BC(Block 2: ADD)  Inputs: 3.0 4.0  Outputs: null,
    BC(Block 2: ADD).out[0].value=null ),
  ( BC(Block 2: ADD).out[0].value=null,
    BC(Block 3: MULT).in(1].value=null ),
  ( BC(Block 3: MULT).in(1].value=null,
    BC(Block 3: MULT) Inputs: null null  Outputs: null )
  ] )

Part 3: Propogate signals: Steps 1 and 2...
-------------------------------------------

Part 4: Meta-Component Graph ...
-----------------------------------
( [
  BC(Block 1: ADD)  Inputs: 1.0 2.0  Outputs: 3.0 ,
  BC(Block 3: MULT)  Inputs: 3.0 7.0  Outputs: 21.0 ,
  BC(Block 3: MULT).in(0].value=3.0,
  BC(Block 1: ADD).out[0].value=3.0,
  BC(Block 2: ADD)  Inputs: 3.0 4.0  Outputs: 7.0 ,
  BC(Block 3: MULT).in(1].value=7.0,
  BC(Block 2: ADD).out[0].value=7.0
  ] , [
  ( BC(Block 1: ADD)  Inputs: 1.0 2.0  Outputs: 3.0,
    BC(Block 1: ADD).out[0].value=3.0),
  ( BC(Block 1: ADD).out[0].value=3.0,
    BC(Block 3: MULT).in(0].value=3.0),
  ( BC(Block 3: MULT).in(0].value=3.0,
    BC(Block 3: MULT)  Inputs: 3.0 7.0  Outputs: 21.0 ),
  ( BC(Block 2: ADD)  Inputs: 3.0 4.0  Outputs: 7.0,
    BC(Block 2: ADD).out[0].value=7.0 ),
  ( BC(Block 2: ADD).out[0].value=7.0,
    BC(Block 3: MULT).in(1].value=7.0 ),
  ( BC(Block 3: MULT).in(1].value=7.0,
    BC(Block 3: MULT)  Inputs: 3.0 7.0  Outputs: 21.0 )
  ] )

BC(Block 1: ADD)  Inputs: 1.0 2.0  Outputs: 3.0
BC(Block 2: ADD)  Inputs: 3.0 4.0  Outputs: 7.0
BC(Block 3: MULT)  Inputs: 3.0 7.0  Outputs: 21.0

Part 5: Result ....
-----------------------------------
Answer: y = 21.0
prompt >>
```

Key points to note:

1. The system architecture is modeled with the seven-node graph structure shown in Figure 12.27. Three nodes are parent components, i.e.,

```
BC(Block 1: ADD)  Inputs:  1.0  2.0  Outputs: null
```



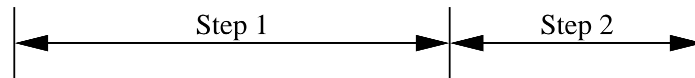


Figure 12.27. Graph structure for the network of arithmetic processors.

```
BC(Block 2: ADD)  Inputs:  3.0  4.0  Outputs: null
BC(Block 3: MULT) Inputs: null null  Outputs: null
```

The remaining four nodes are input and output ports, i.e.,

```
BC(Block 1: ADD).out[0].value=null,
BC(Block 2: ADD).out[0].value=null,
BC(Block 3: MULT).in(0].value=null,
BC(Block 3: MULT).in(1].value=null,
```

Four graph edges correspond to links between the input/output ports and their parent components, i.e.,

```
( BC(Block 1: ADD) Inputs: 1.0 2.0 Outputs: null,
  BC(Block 1: ADD).out[0].value=null ),
( BC(Block 2: ADD) Inputs: 3.0 4.0 Outputs: null,
  BC(Block 2: ADD).out[0].value=null ),
( BC(Block 3: MULT).in(0].value=null,
  BC(Block 3: MULT) Inputs: null null Outputs: null ),
( BC(Block 3: MULT).in(1].value=null,
  BC(Block 3: MULT) Inputs: null null Outputs: null )
```

The remain two edges connect blocks 1 and 2 to block 3, i.e.,

```
( BC(Block 1: ADD).out[0].value=null, BC(Block 3: MULT).in(0].value=null ),
( BC(Block 2: ADD).out[0].value=null, BC(Block 3: MULT).in(1].value=null ),
```

2. At the beginning of the data processing, the contents of blocks 1 through 3 are:

```
BC(Block 1: ADD)   Inputs:  1.0  2.0  Outputs: null
BC(Block 2: ADD)   Inputs:  3.0  4.0  Outputs: null
BC(Block 3: MULT)  Inputs: null null  Outputs: null
```

And at the end they are:

```
BC(Block 1: ADD)   Inputs: 1.0 2.0  Outputs:  3.0
BC(Block 2: ADD)   Inputs: 3.0 4.0  Outputs:  7.0
BC(Block 3: MULT)  Inputs: 3.0 7.0  Outputs: 21.0
```

The first call to

```
manager.process()
```

systematically evaluates the block functionality for all of the JGraphT nodes which implement the Component interface. For our simple application, blocks 1 and 2 have data values on their input ports – the corresponding addition operations are computed and propogated across the wire to the input ports for block 3. Block 3 has functionality, but during step one neither of the input ports have been initialized with data, so nothing happens.

During the second call to `manager.process()`, the arithmetic operations for blocks 1 and 2 are repeated – obviously very inefficient – and the arithmetic operation for block 3 is computed.

Example 08. Hierarchy of Components for Data-Flow Processing

Now, let us implement a base component:

source code

```
/**
 *  ==================================================================================
 *  MetaComponent.java
 *  ==================================================================================
 */

package component;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;

public class MetaComponent<T> implements Component<T> {
   private String sName;

   public void setName ( String sName ) {
      this.sName = sName;
   }

   public interface MetaVisitor<T> {
      public void visit(Component<T> component);
   }

   private HashMap<OutputPort<T>,WireImpl<T>> sourceWireMap;
   private ArrayList<Wire<T>> wires;
   private HashSet<Component<T>>  subComponents;
   private ArrayList<InputPort<T>>  externalInputs  = new ArrayList<InputPort<T>>();
   private ArrayList<OutputPort<T>> externalOutputs = new ArrayList<OutputPort<T>>();

   public void accept(MetaVisitor<T> v) {
      for (Component<T> item : subComponents) {
          v.visit(item);
      }
   }

   public int getInputSize() {
      return externalInputs.size();
   }

   public int getOutputSize() {
      return externalOutputs.size();
   }

   public InputPort<T> getInputPort(int index) {
      return externalInputs.get(index);
   }

   public OutputPort<T> getOutputPort(int index) {
      return externalOutputs.get(index);
   }
```

```
public MetaComponent() {
   subComponents = new HashSet<Component<T>>();
   wires = new ArrayList<Wire<T>>();
   sourceWireMap = new HashMap<OutputPort<T>,WireImpl<T>>();
}

public void addConnection(OutputPort<T> src, InputPort<T>... targets) {
   for (InputPort<T> target : targets) {
        addConnection(src, target);
   }
}

public void addConnection(OutputPort<T> src, InputPort<T> target) {

   // Add ports' parent components to my subcomponent list

   subComponents.add(src.getParent());
   subComponents.add(target.getParent());
   WireImpl<T> srcWire = sourceWireMap.get(src);

   if (srcWire == null) {

       // make a new wire

       srcWire = new WireImpl<T>(src);
       sourceWireMap.put(src, srcWire);
   }

   // TODO: Remove any old wire connections using the target?
   // add the target port to the wire

   srcWire.addTargetPort(target);
}

private void propagateSignals() {
   for (Wire<T> w : wires) {
      w.propagateSignal();
   }
}

private void processSubComponents() {
   for (Component<T> item : subComponents) {
      item.process();
   }
}

public void process() {
   processSubComponents();
   propagateSignals();
}

public void processRepeat(int count) {
   if (count <= 0) {
      return;
```

```
      }

      for (int i=0; i<count; i++) {
         process();
      }
   }

   public void addExternalPort(InputPort<T> port) {
      subComponents.add(port.getParent());
      externalInputs.add(port);
   }

   public void addExternalPort(OutputPort<T> port) {
      subComponents.add(port.getParent());
      externalOutputs.add(port);
   }
}
```

and build an application ...

source code

```
TBD ...
```

Bibliography

[1] Alexander C., Ishikawa S., and Silverstein M. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[2] Gamma, E., Helm R., Johnson, R., and Vlissides, J. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] Kjolaas K.A.H. Automatic Furniture Population of Large Architectural Models. *M.S. Thesis, Department of Electrical Engineering and Computer Science, MIT*, 2000.

[4] Stelting S. and Maassen O. *Applied Java Patterns*. The SUN Microsystems Press/Prentice-Hall, 2002.