ENCE 688P

Abstract Classes and Interfaces

Abstract Classes and Interfaces

Part 1. Framework for Component-Based Design

- Framework for design reuse, enabled by software interfaces.

Part 2. Working with Abstract Classes

- Definition and Implementation

- Examples: Efficient modeling of shapes; class hierarchy for a retail catalog.

Part 3. Working with Interfaces, Abstract Classes and Interfaces

- Motivation and implementation.

- Example: Software interfaces for farm workers.

- Programming to an Interface

Part 4. Applications

- State design pattern; evaluation of functions with JEval; interface specification for a spreadsheet; class diagram hierarchy and modeling for an interconnect system.
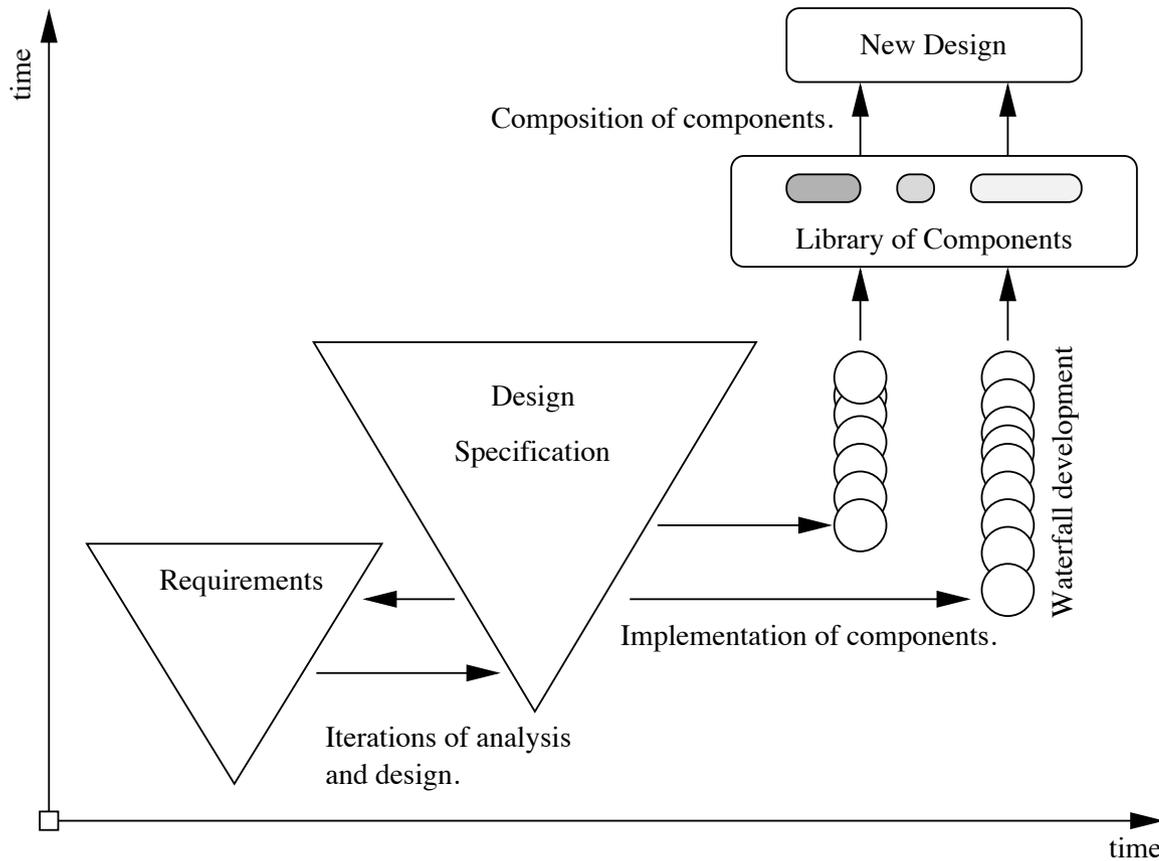
Part 1. Framework for Component-Based Design

Component-Based Development

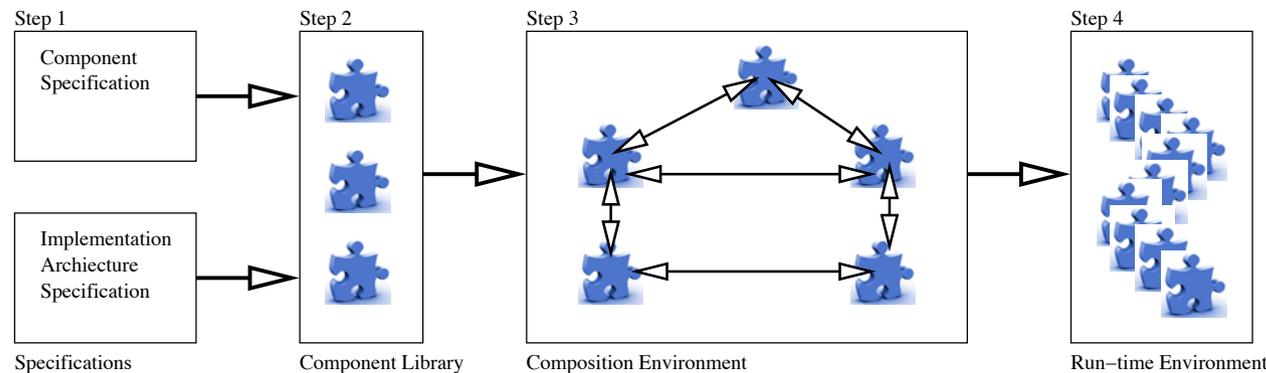**Pathway of Development for Reuse-Focused Design**

Component-Based Development

Preliminary Observations for Reuse-Focused Design

- Component-based system development efforts are motivated by the need to keep ever-increasing system complexity in check, to reduce system delivery times, improve consistency, improve visibility, and provide support for parallel and distributed development.

- In a departure from the goals of object-oriented system development, ...

    ... component-based system development is primarily concerned with the design and assembly of solutions as a collection of interacting pieces.

Simplified View of a Component Technology Supply Chain
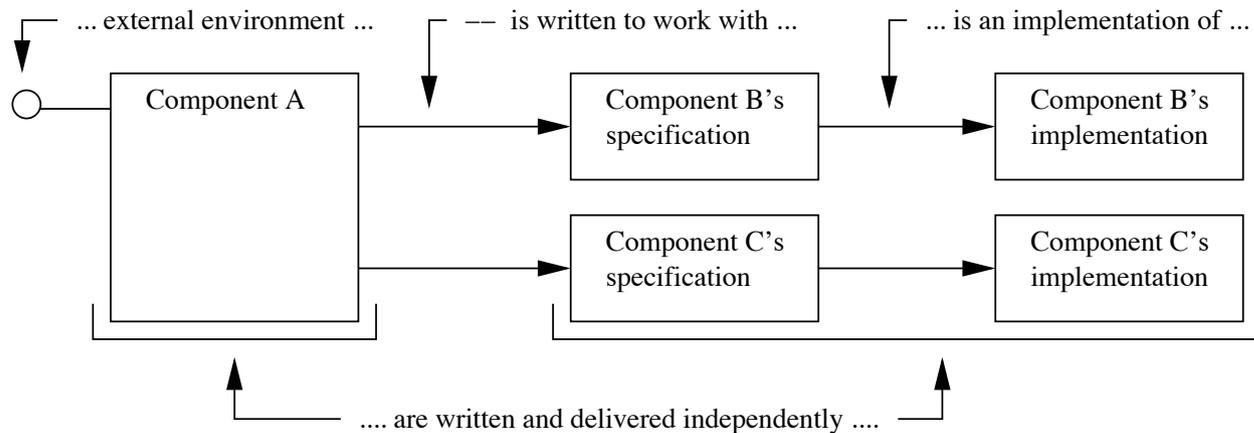

Step 1	Step 2	Step 3	Step 4
Component Specification			
Implementation Archiecture Specification			
Specifications	Component Library	Composition Environment	Run–time Environment

Component-Based Development

Schematic of a Simple Component-Based Software System

Implementation requires ...

... techniques for describing the overall system architecture, and for the definition of pieces in a way that facilitates assembly with other pieces.

Component-Specification-Implementation Pathway

```
      ... external environment ...      ── is written to work with ...      ... is an implementation of ...

   ○──┌──────────────┐      ┌──────────────────┐        ┌──────────────────┐
      │  Component A  │─────▶│  Component B's    │───────▶│  Component B's    │
      │              │      │  specification    │        │  implementation   │
      │              │      └──────────────────┘        └──────────────────┘
      │              │      ┌──────────────────┐        ┌──────────────────┐
      │              │─────▶│  Component C's    │───────▶│  Component C's    │
      └──────────────┘      │  specification    │        │  implementation   │
                            └──────────────────┘        └──────────────────┘

              └──────▲──────┘        └──────────────▲──────────────┘
                     └──── .... are written and delivered independently .... ────┘
```

Components B and C are defined via their specifications/interfaces. Component A employs the services of compoments B and C.

Interface-Based Development

Pathway from Component- to Interface-Based Design

- During the early stages of design where the focus is on understanding the roles and responsibilities of components within a domain, ...

  ... interfaces play the primary role in descision making for what the implemented system might look like.

- This gives rise to the term interface-based design.

- Key Benefit: Experience indicates that:

  ... focusing on interfaces as the key design abstraction leads to much more flexible designs.

Remark. Interface-based design procedures are particularly important for the design and managed evolution of systems-of-systems.

Part 2. Abstract Classes

Working with Abstract Classes

Definition

Abstract classes provide an abstract view of a real-world entity or concept.

> **They are an ideal mechanism when you want to create something for objects that are closely related in a hierarchy.**

Implementation

- An abstract class is a class that is declared abstract. It may or may not include abstract methods.

- Abstract classes cannot be instantiated (i.e., you cannot create an object from an abstract class). But they can be subclassed.

- When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class.

Working with Abstract Classes

Example 1. Efficient Modeling of Shapes

In this example we ....

> **... model shapes under the single umbrella of a Shapes class, and then gain computational efficiencies by organizing the implementation of all shapes into a single common hierarchy.**

Definition

A shape is a

> **... high-level geometric concept that can be specialized into specific and well-known two-dimensional geometric entities.**
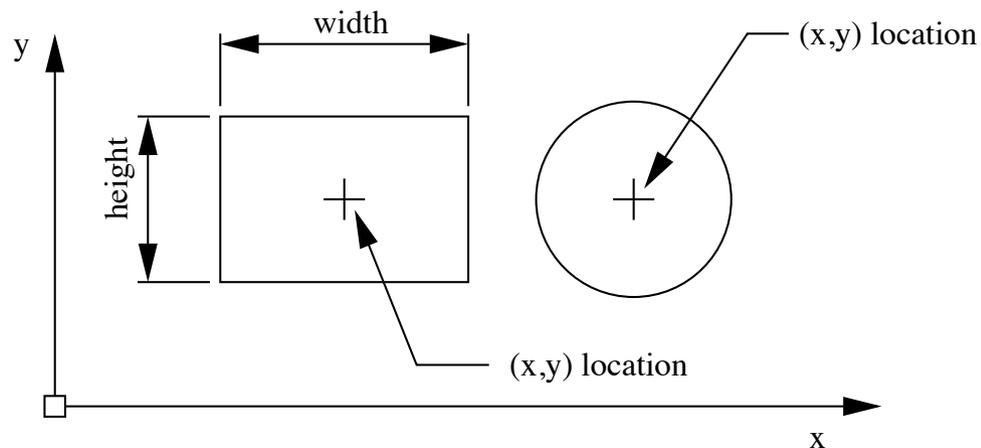
Examples: ovals, circles, rectangles, triangles, octogons, and so forth.

Working with Abstract Classes

Capturing Shape Data

There are ...

... sets of data values and computable properties that are common to all shapes.

For example, shapes have an area, perimeter, an (x,y) centroid and a position or (x,y) location.

Working with Abstract Classes
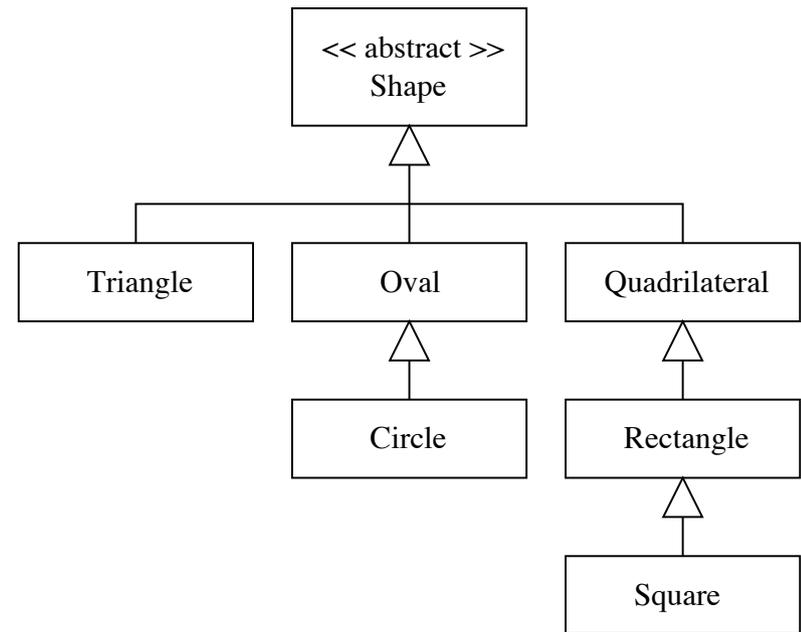
Organization of Shapes into a Hierarchy

Specific types of shapes can be ...

   **... organized into a natural hierarchy.**

Examples

- Squares are a specific type of rectangle, which in turn, are a specific type of quadralateral.

- Circles can be viewed as a special type of oval.

Many other shapes are possible: point, linesegment, rhombus, parallelogram, kite, ..etc.

```
                    +-------------+
                    | << abstract >> |
                    |    Shape    |
                    +-------------+
                          △
        +-----------------+-----------------+
        |                 |                 |
   +----------+      +----------+      +--------------+
   | Triangle |      |   Oval   |      | Quadrilateral|
   +----------+      +----------+      +--------------+
                          △                   △
                     +----------+      +--------------+
                     |  Circle  |      |  Rectangle   |
                     +----------+      +--------------+
                                              △
                                       +--------------+
                                       |   Square     |
                                       +--------------+
```

Working with Abstract Classes

Class Diagram for TestShape Program
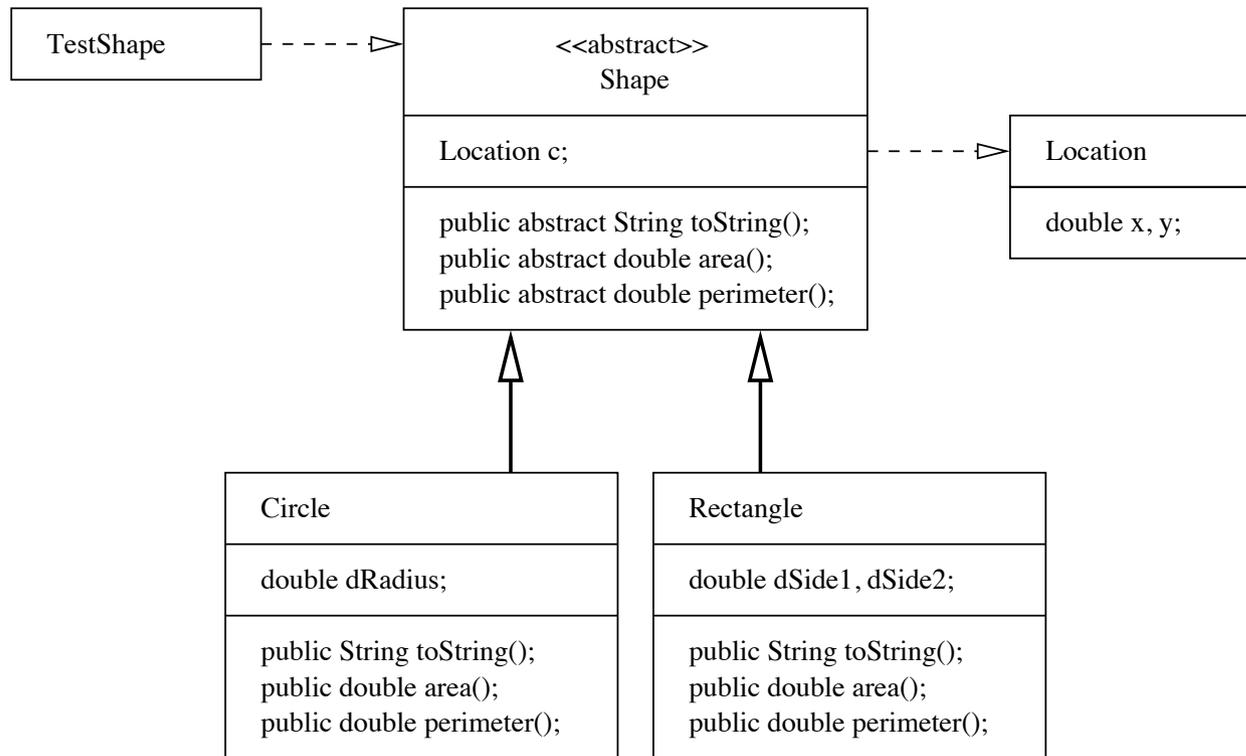
All extensions of Shape will need to provide concrete implementations for the methods area(), perimeter() and toString().

Working with Abstract Classes

Implementation Efficiency and Convenience

- Instead of solving problems with algorithms that work with specific object types (e.g., Rectangles and Circles), algorithms can be developed for shapes.

```
Shape s[] = new Shape [3] ;

s[0] = new Rectangle( 3.0, 3.0, 2.0, 2.0 );
s[1] = new Circle( 1.0, 2.0, 2.0 );
s[2] = new Rectangle( 2.5, 2.5, 2.0, 2.0 );
```

  The JVM will figure out the appropriate object type at run time.

- Use of the abstract shape class reduces the number of dependencies in the program architecture.

  Thus, from a systems standpoint, ...

  ... the program architecture is loosely coupled and ammenable to change.

  For example, it would be a trivial matter to add Triangles to the class hierarchy.

Working with Abstract Classes

Walking Along an Array of Shapes

```
System.out.println("--------------------");
for (int ii = 1; ii <= s.length; ii = ii + 1) {
    System.out.println(  s[ii-1].toString() );
    System.out.println( "Perimeter = " + s[ii-1].perimeter() );
    System.out.println("--------------------");
}
```

Program Output

```
prompt >>
--------------------
Rectangle : Side1 = 3.0 Side2 = 3.0
Perimeter = 12.0
--------------------
Circle : Radius = 1.0 [x,y] = [2.0,2.0]
Perimeter = 6.283185307179586
--------------------
Rectangle : Side1 = 2.5 Side2 = 2.5
Perimeter = 10.0
--------------------
prompt >>
```

Working with Abstract Classes

Example 2. Class Diagram for Operation of a Retail Catalog
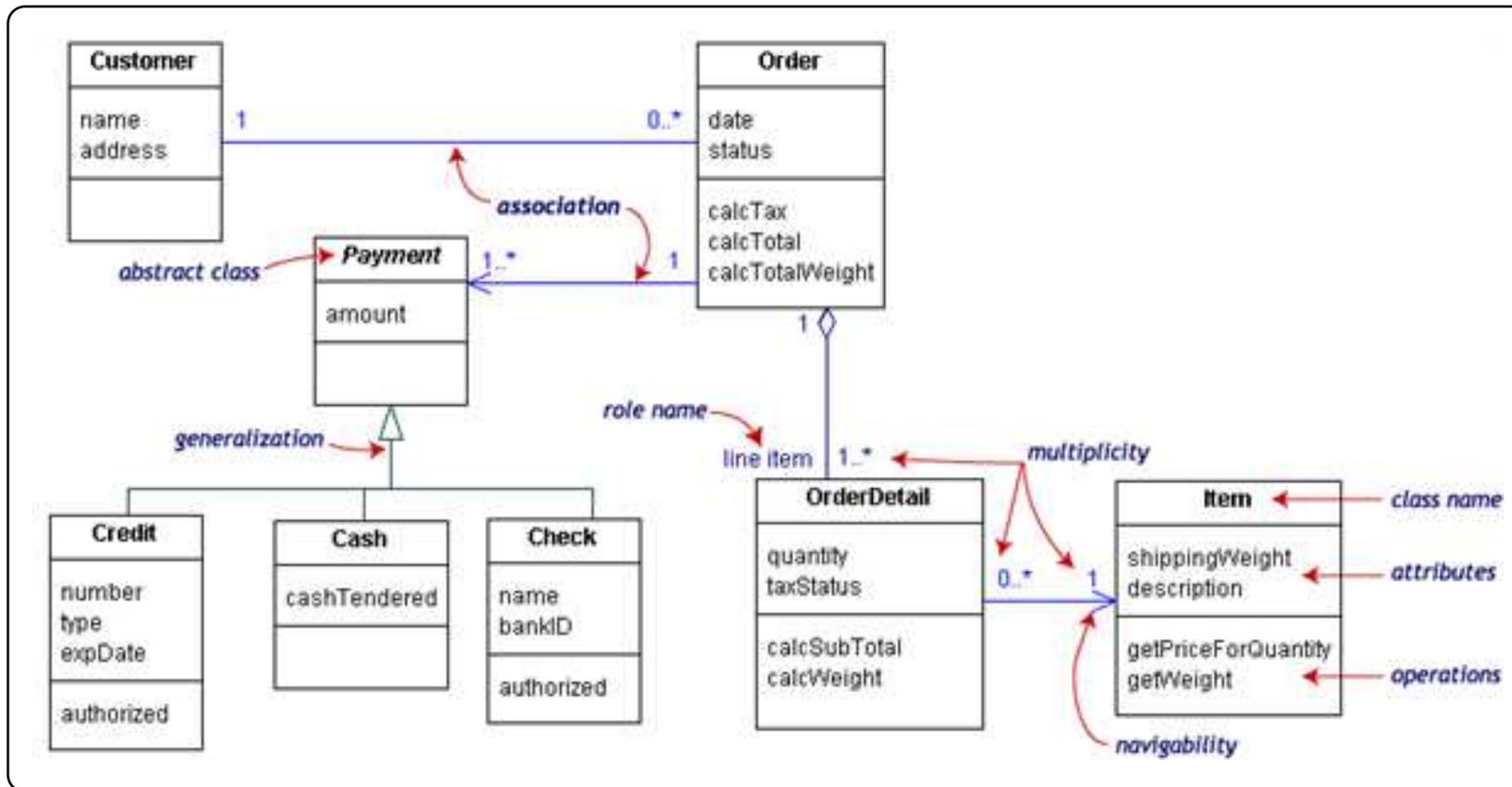
Working with Abstract Classes

Points to Note:

This example conveys the following messages:

- The central class is the Order.

- Associated with each order are the Customer making the purchase and the Payment.

- Payments is an abstract generalization for: Cash, Check, or Credit.

- The order contains OrderDetails (line items), each with its associated Item.

Also note:

- UML class notation is a rectangle divided into three parts: class name, attributes, and operations.

- Names of abstract classes, such as Payment, are in italics.

- Relationships between classes are the connecting links.

Part 3. Working with Interfaces

Working with System Interfaces

Motivation

Interfaces are the mechanism by which ...

> **... components describe what they do (or provide in terms of functionality and/or services).**

Interface abstractions are appropriate for collections of objects that provide:

> **... common functionality, but are otherwise unrelated.**

Implementation

- An interface defines a set of methods without providing an implementation for them.

- An interface does not have a constructor – therefore, it cannot be instantiated as a concrete object.

- Any concrete class the implements the interface must provide implementations for all of the methods listed in the interface.

Example 1. Software Interface for Farm Workers

Class diagram for implementation and use of a farm workers interface.
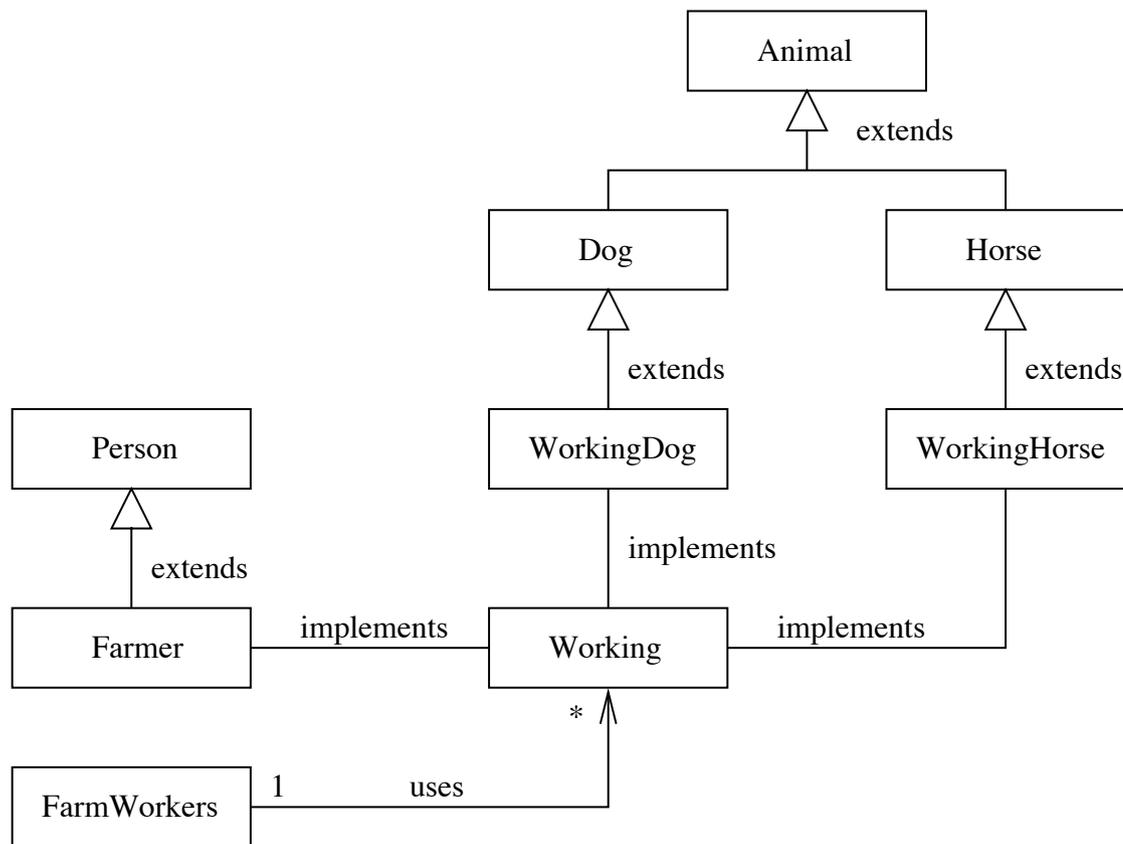
Working with System Interfaces

Example 1. Software Interface for Farm Workers

Workers is simply an abstract class that defines an interface, i.e.,

```
public interface Working {
    public abstract void hours ();
}
```

In Java, the interface is implemented by using the keyword "implements" in the class declaration, e.g.,

```
public class Farmer implements Working { ....
```

This declaration ...

... sets up a contract that guarantees the Farmer class will provide a concrete implementation for the method hours().

Working with System Interfaces

Important Point

Instead of writing code that looks like:

```
Farmer       mac = new Farmer (...);
WorkingDog   max = new WorkingDog (...);
WorkingHorse silver = new WorkingHorse (...);
```

We can treat this group of objects as a set of Working entities, i.e.,

```
Working    mac = new Farmer (...);
Working    max = new WorkingDog (...);
Working silver = new WorkingHorse (...);
```

Methods and algorithms can be defined in terms of all "Working" entities, independent of the lower-level details of implementation.

Programming to an Interface

Motivation and Benefits

In Java, an interface represents ...

... what a class can do, but not how it will do it, which is the actual implementation.

Two key benefits:

- Information hiding – as long as the objects conform to the interface specification, then there is no need for the clients to know the exact type of the objects they use.

- Improved flexibity – system behavior can be changed by swapping the object used with another implementing the same interface.

Programming to an Interface

Combining Abstract Classes and Interfaces

```
                                    implements
   +------------------+   - - - - - - - - - - - - -->  +------------------+
   |   << abstract >> |                                |  << Interface >> |
   |        A         |                                |        B         |
   +------------------+                                | method1()        |
   | method1()        |                                | method2()        |
   | method2()        |                                | method3()        |
   | method3()        |                                +------------------+
   +------------------+
              △
     +--------+--------+
     |        |        |
+---------+ +---------+ +---------+
|    C    | |    D    | |    E    |
| method1()| method1()| method3()|
| method2()| method2()|         |
| method3()|         |         |
+---------+ +---------+ +---------+
```
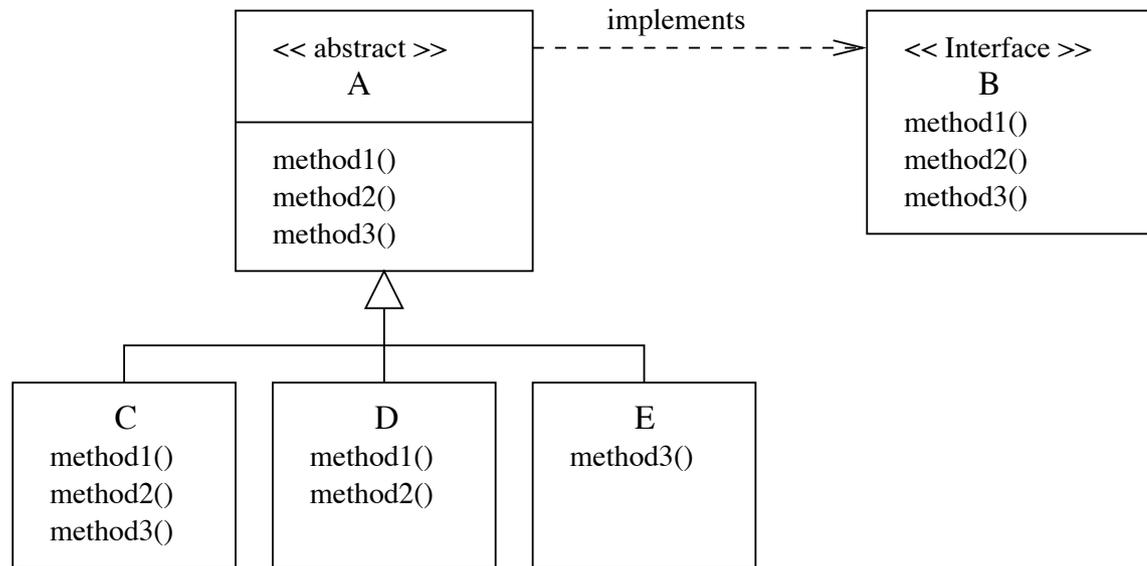
Now we can write:

```
Creating objects of type C,D and E.   Executing methods ...

====================================== ======================
B c1 = new C (...);                    c1.method1();

B d1 = new D (...);                    d1.method2();

B e1 = new E (...);                    e1.method3();

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

Programming to an Interface

Example. Two Factories making Widgets
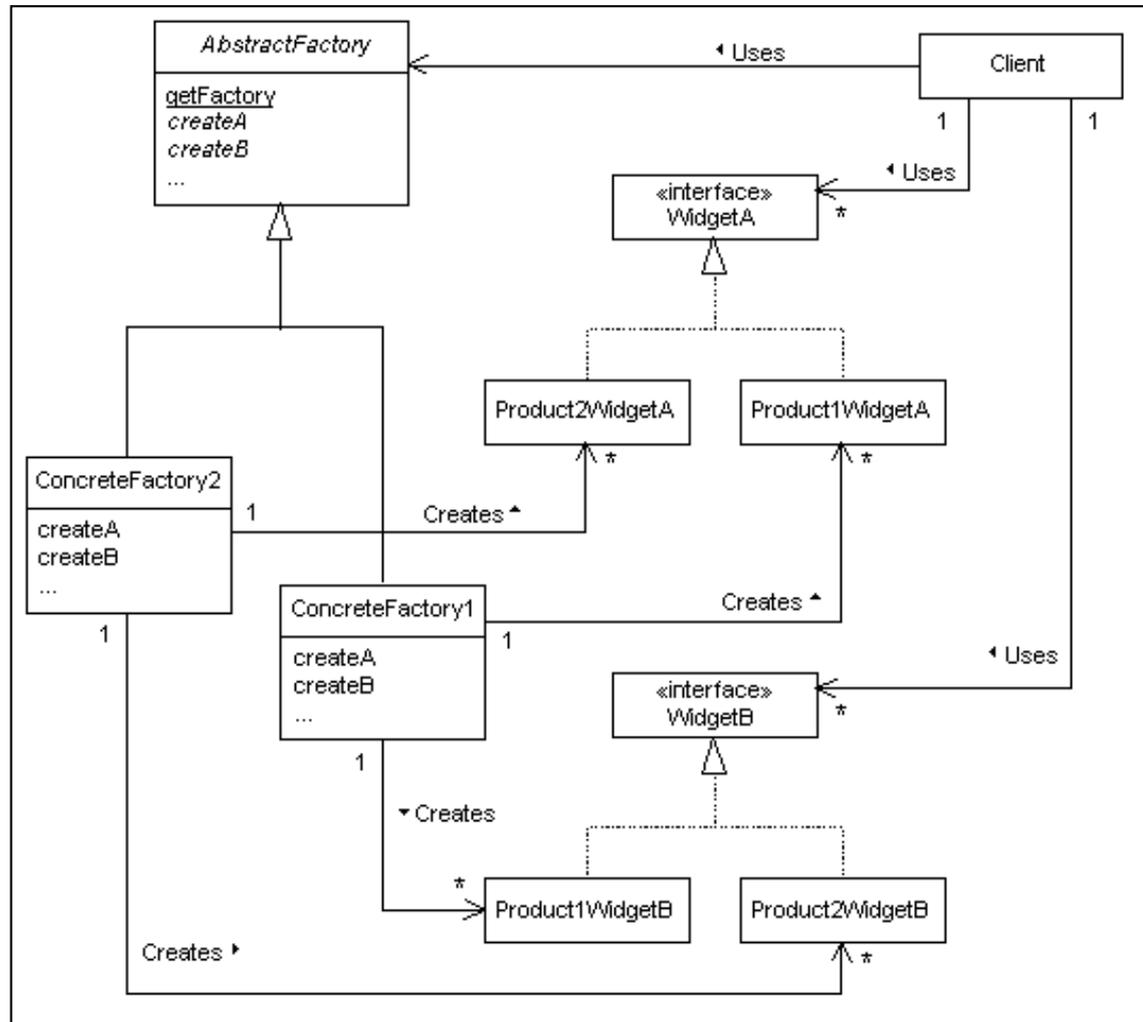
Abstract Classes and Interfaces

Part 4. Applications

Application: State Design Pattern

Application 1. State Design Pattern (pg. 106 of Stelting)

Purpose

- To easily change an object's behavior at runtime.

Description

- The state design patterns allows for the dynamic real-time adjustment of object behavior.

- It represents the states of an object as discrete objects.

Implementation

- Dynamic behavior is achieved by delegating all method calls that certain values of to a State object (i.e., to the system's current state).

- In this way, the implementation of those methods can vary with the state of the object.

- No need for lengthy if-else statements.

Application: State Design Pattern

Class Hierarchy for Implementation
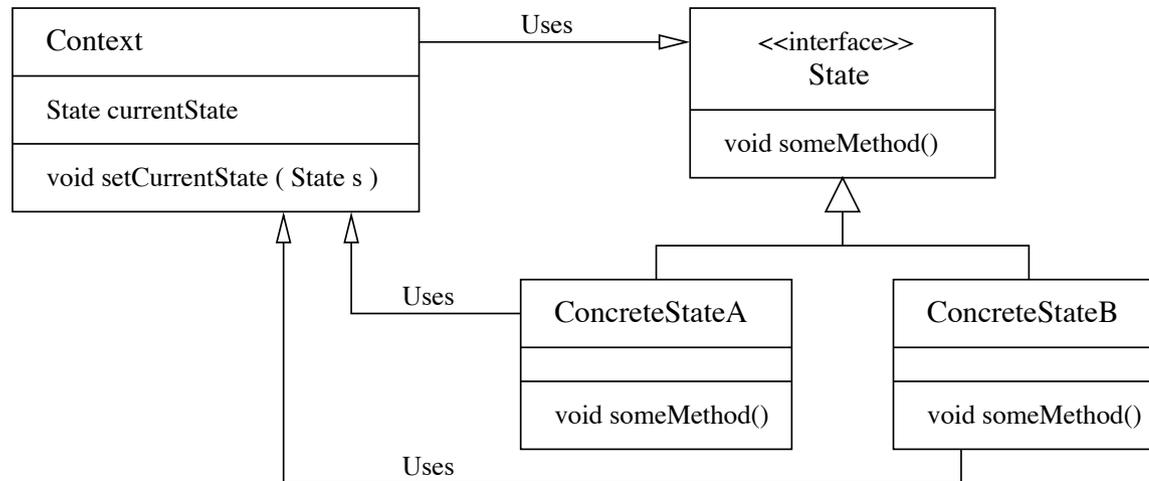
Implementation of the state design pattern requires:

- A `Context` object that keeps reference to the current state. State-specific method calls are delegated to the current state object.

- A `State` interface that defines all of the methods that depend on the state of the object.

- A family of `ContreteState` objects.

Application: State Design Pattern

Application. Toggle Behavior for a Simple Button

State behavior can be summarized as follows:

- If the system state is ON and the button is pushed, then the system will transition to an OFF state, and

- If the system state is OFF and the button is pushed, then the system will transition to an ON state.

Here is the State interface:

```
public interface State { public void push( Button b ); }
```

and here is the Button class:

```
public class Button {
    private State current;
    public Button()                     { current = OFF.instance(); }
    public void setCurrent( State s ) { current = s; }
    public void push()                  { current.push( this ); }
}
```

Application: State Design Pattern

Application. Toggle Behavior for a Simple Button

Here is ToggleButton.java:

```
public class ToggleButton {
    public static void main( String[] args ) {
        Button power = new Button();
        for ( int i = 1; i <= 5; i = i + 1 )
            power.push();
    }
}
```

The program output is as follows:

```
prompt >> java ToggleButton
    button: turning ON
    button: turning OFF
    button: turning ON
    button: turning OFF
    button: turning ON
prompt >>
```

Here is ON.java

```
public class ON implements State {
    private static  ON inst = new ON();
    private ON() { }

    public static State instance() {
        return inst;
    }

    public void push( Button b ) {
        b.setCurrent( OFF.instance() );
        System.out.println(
            " button: turning OFF" );
    }
}
```

Application: Evaluation of Functions with JEval

Application 2: Parsing and Evaluation of Functions with JEval

JEval is the advanced library for adding mathematical, string, Boolean and functional expression parsing and evaluation to your Java applications.

Summary of features:

- Parses and evaluates dynamic and static expressions at run time.

- A great solution for filtering data at runtime.

- Supports mathematical, Boolean, String and functional expressions.

- Supports all major mathematical and Boolean operators.

- Supports custom functions.

- 39 Math and String functions built in and ready to use.

- Supports variables and nested functions.

Application: Evaluation of Functions with JEval

Examples: Relational and Arithmetic Expressions

- String sExp = "(2 < 3) || ((1 == 1) && (3 < 3))";

- String sExp = "1 + 2 + 3*4 + 10.0/2.5";

- String sExp = "1 + abs(-1)";

- String sExp = "atan2(atan2(1, 1), 1)";

- String sExp = "acos(-1.0)";

Examples: Working with Strings

- String sExp = "toLowerCase('Hello World!')";

- String sExp = "toUpperCase(trim(trim(' a b c ')))";

Application: Evaluation of Functions with JEval

Examples: Working with variables

```
String sEexp = "#{a} >= 2 && #{b} >= 5 && #{c} >= 8";

Long a = (Long) row.get(0);
evaluator.putVariable("a", a.toString());
Long b = (Long) row.get(1);
evaluator.putVariable("b", a.toString());
Long c = (Long) row.get(2);
evaluator.putVariable("c", a.toString());

... etc ...

String result01 = evaluator.evaluate( sExp );
```

Application: Evaluation of Functions with JEval

Builtin String Functions

CharAt.java	CompareTo.java	Concat.java	EndsWith.java	Equals.java
Eval.java	IndexOf.java	LastIndexOf.java	Length.java	Replace.java
StartsWith.java	Substring.java	ToLowerCase.java	ToUpperCase.java	Trim.java

Builtin Math Functions

Abs.java	Acos.java	Asin.java	Atan.java	Atan2.java
Ceil.java	Cos.java	Exp.java	Floor.java	Log.java
Max.java	Min.java	Pow.java	Random.java	Rint.java
Round.java	Sin.java	Sqrt.java	Tan.java	ToDegrees.java
ToRadians.java				

Builtin Operator Functions

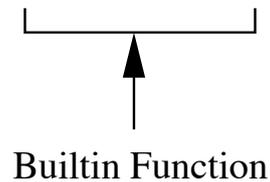|		
AbstractOperator.java	DivisionOperator.java	ModulusOperator.java
AdditionOperator.java	EqualOperator.java	MultiplicationOperator.java
BooleanAndOperator.java	GreaterThanOperator.java	NotEqualOperator.java
BooleanNotOperator.java	GreaterThanOrEqualOperator.java	OpenParenthesesOperator.java
BooleanOrOperator.java	LessThanOperator.java	Operator.java
ClosedParenthesesOperator.java	LessThanOrEqualOperator.java	SubtractionOperator.java

Application: Evaluation of Functions with JEval

Syntax and Semantics

String sEexp = " #{ a } >=   2 && # { b } >= 6 && #{ c } >= 8 }";

Variable a   Variable b

Logical And Operator

Greater than or equal to Operator
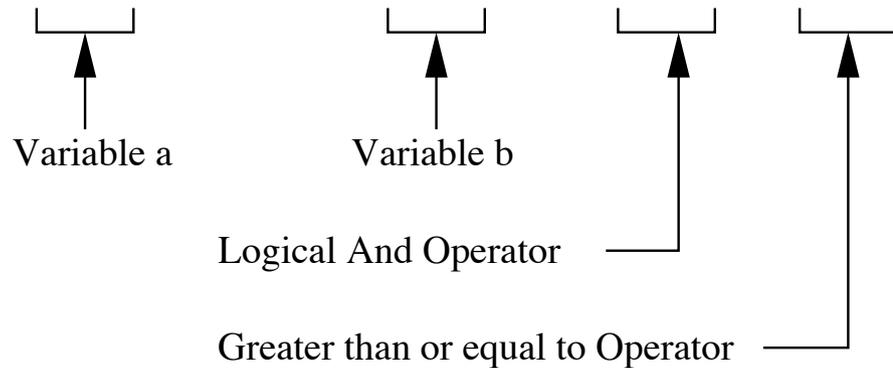
String sEexp = " atan2 (atan2 (1, 1), 1)";

Builtin Function

Application: Evaluation of Functions with JEval

Function Interface

```java
public interface Function {

    // Return name of the function ...

    public String getName();

    // Execute the function for a specified argument ...

    public FunctionResult execute(Evaluator evaluator, String arguments) ...
}
```

Using the Function Interface

```java
public class Acos implements Function { ... } ....
public class Max  implements Function { ... } ....
```

Application: Evaluation of Functions with JEval

Operator Interface

```
public interface Operator {
    // Evaluates two double operands.

    public abstract double evaluate(double leftOperand, double rightOperand);

    // Evaluate one double operand ...

    public abstract double evaluate(final double operand); ....
}
```

Using the Operator Interface

```
public abstract class AbstractOperator implements Operator { ... }

public class DivisionOperator extends AbstractOperator { ... }
public class BooleanAndOperator extends AbstractOperator { ... }
```

Application: Using Interfaces in Spreadsheets

Application 3: Graphical Interface
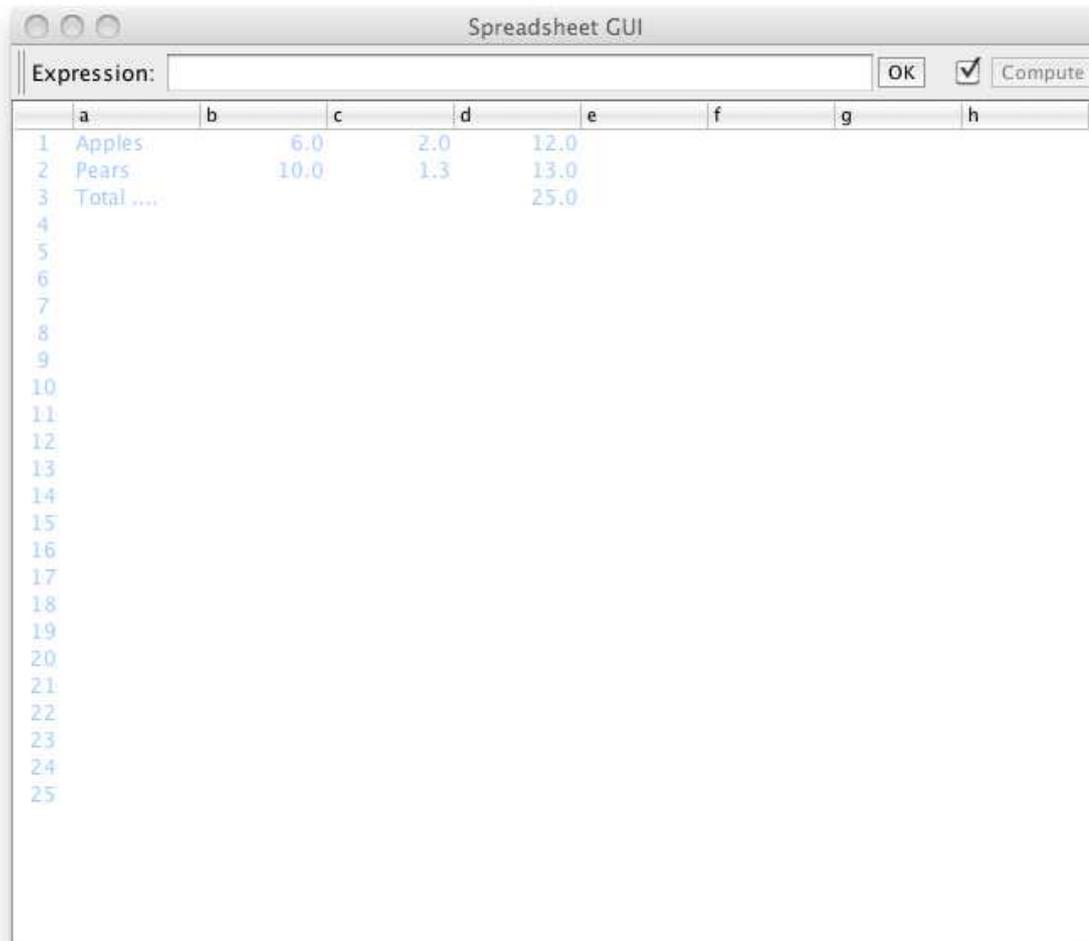
Application: Using Interfaces in Spreadsheets

Modeling a Spreadsheet Cell

```
public class Cell {
   private String expression;    // expression in cell
   private Set<String> children; // list of cells which reference this
   private Set<String> parent;   // list of cells this references
   private Object value;         // Value of displayed cell ...

   // Class constructor

   public Cell() {
      children = new TreeSet<String>();
      parent   = new TreeSet<String>();
   }

   ..... etc .....
}
```

Application: Using Interfaces in Spreadsheets

Basic Spreadsheet Interface

```java
public interface SpreadsheetInterface {
   public static final String LOOP = "#LOOP"; // loop Error Value
   public int getColumnCount();                // Number of columns in the spreadsheet.
   public int getRowCount();                   // Number of rows in the spreadsheet.

   // Set and get the cell expression at prescribed location...

   public   void setExpression(String location, String expression);
   public String getExpression(String location);

   // Returns the expression stored at the cell at location.

   public Object getValue(String location);

   // Returns the value associated with the computed stored expression.

   public void recompute();
}
```

Application: Using Interfaces in Spreadsheets

Extended Spreadsheet Interface

```java
public interface IterableSpreadsheetInterface extends SpreadsheetInterface {

    // Set/get the number of times to compute the value stored in each loop cell.

    public void setMaximumIterations(int maxIterationCount);
    public  int getMaximumIterations();

    // Set/get the maximum change in value between successive loop iterations...

    public   void setMaximumChange(double epsilon);
    public double getMaximumChange();

    // Recompute value of all cells ...

    public void recomputeWithIteration();
}
```

Application: Using Interfaces in Spreadsheets

Creating the Spreadsheet Model

```
public class Spreadsheet implements SpreadsheetInterface {
    private int numRows, numColumns;      // no. of rows and cols for spreadsheet
    private Map<String, Cell> cells;      // collection of all cells in spreadsheet
    private String lastCellLocation;      // stores location of last cell accessed

    // Set expression of the cell at location ...

    public void setExpression(String location, String expression) { ... }

    // Recompute value of all cells ....

    public void recompute() { ... }

    // Use DFS to check for loops in the relationships among cells ...

    private void checkLOOP(String cellLocation) { ... }
}
```

Application: Using Interfaces in Spreadsheets

Creating a Spreadsheet Object

```
int columns = Integer.parseInt(args[0]);
int rows    = Integer.parseInt(args[1]);

final SpreadsheetInterface spreadsheet = new Spreadsheet(rows, columns);

javax.swing.SwingUtilities.invokeLater(new Runnable() {
   public void run() {
      new SpreadsheetGUI("Spreadsheet GUI", spreadsheet);
   }
});
```

Application: Architecture for Interconnect System

Problem Statement. Hierarchy and network abstractions in a two-layer component/container model.
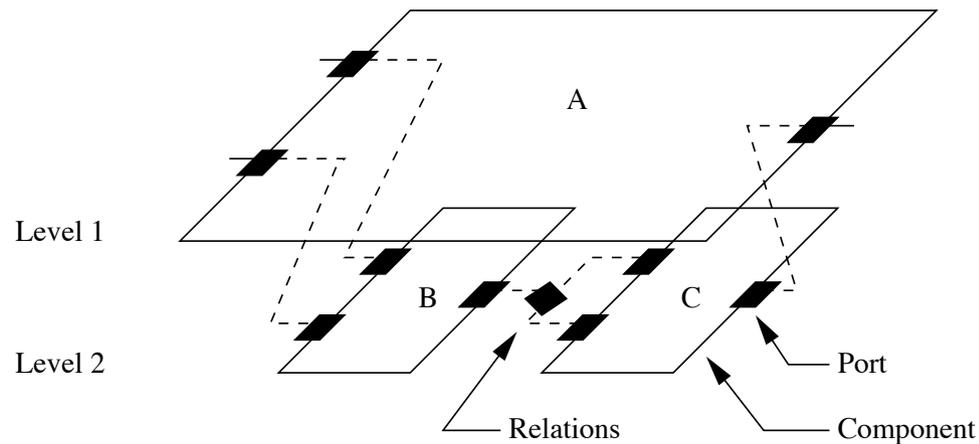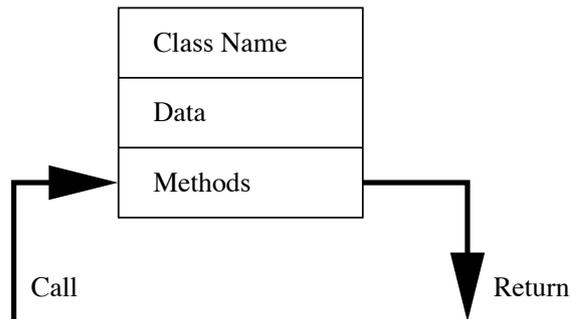

Organizational constraints:

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

- A port cannot be contained by more than one entity. Links cannot cross levels in the hierarchy,

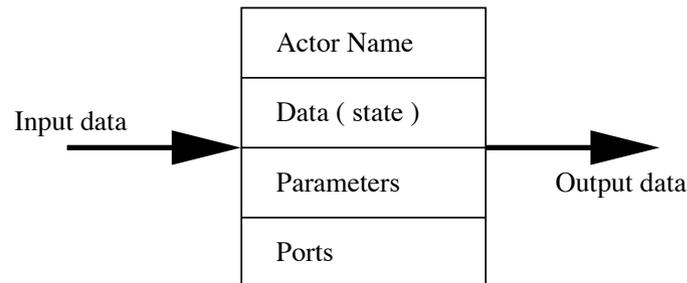- Port-to-port communications must have compatible data types (e.g., signal, energy).

Application: Architecture for Interconnect System

Actor-Oriented Models and Design (adapted from Lee, 2003)

Object–Oriented Design

Class Name
Data
Methods

Call → Return

Actor–Oriented Design

Actor Name
Data (state)
Parameters
Ports

Input data → → Output data

**Object-Oriented Modeling and Design**

- Components interact primarily through method calls (transfer of control).

Actor-Oriented Modeling and Design

- Components interact via some sort of messaging scheme that is **typically concurrent**.

- Constraints in the flow of control define the model of computation.

- Rules define what an actor does (e.g. perform external communication) and when.
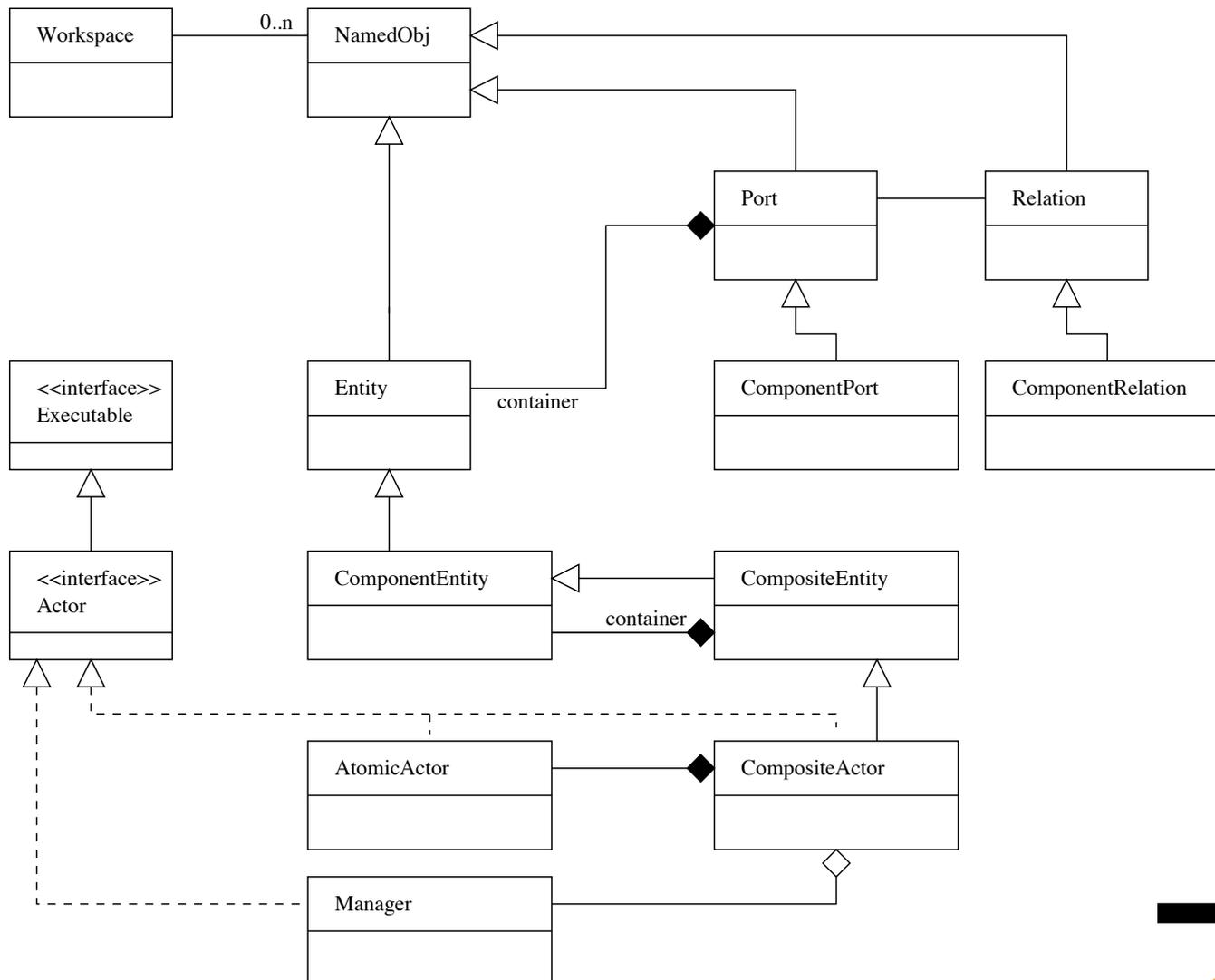
Application: Architecture for Interconnect System

Typical Ptolemy Application (see Brooks et al., 2008)
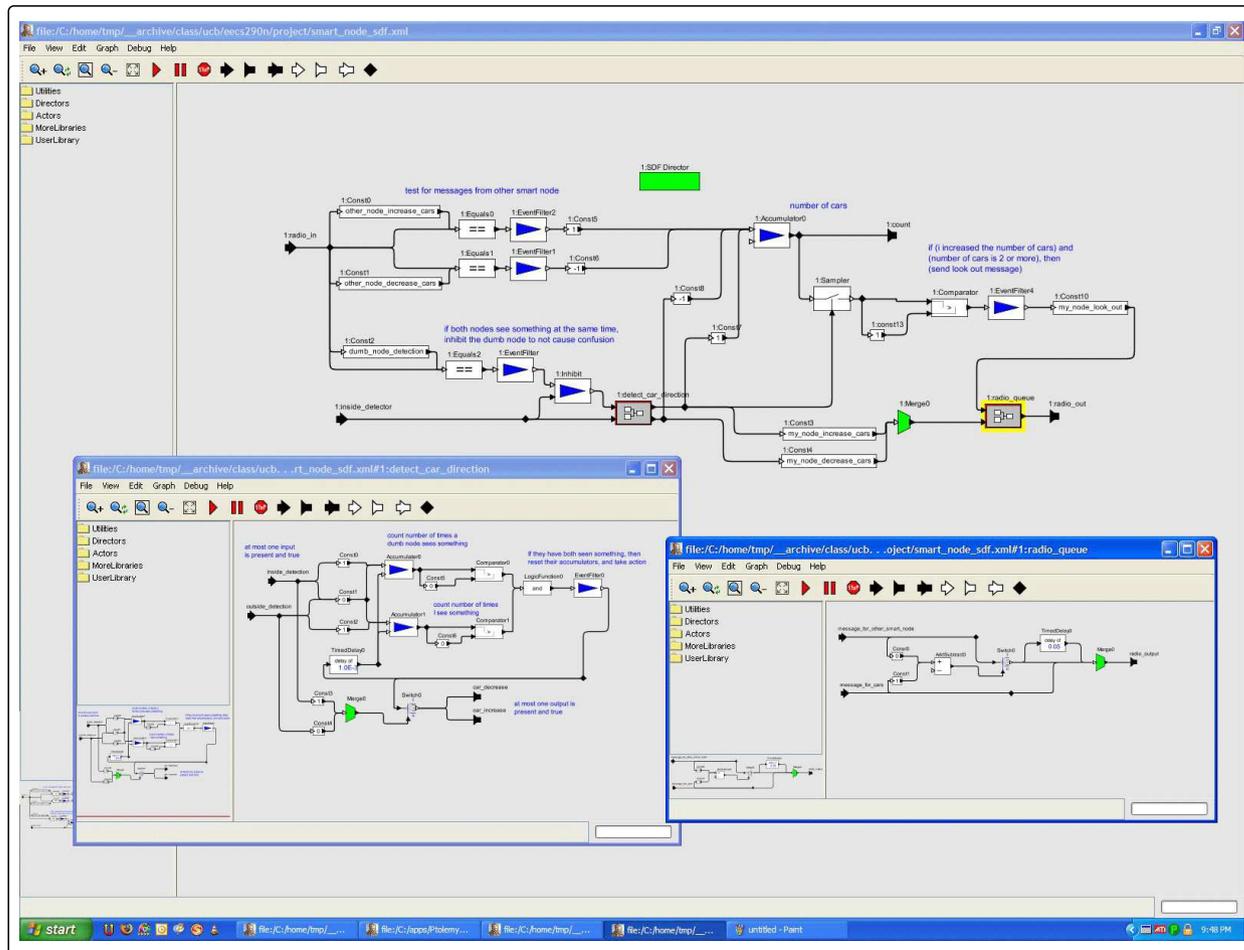
Application: Architecture for Interconnect System

Abbreviated class diagram for modeling of system architectures in Ptolemy.

Application: Architecture for Interconnect System

From Individual Components to Networks of Components

Networks of components form graphs:

- **Graph.** A graph is an object that contains nodes and edges. Edges are accessed through the nodes that they connect.

- **Node.** A node is an object that is contained by a graph and is connected to other nodes by edges.

  A node has a semantic object that is its semantic equivalent in the application and may have a visual object which is its syntactic representation in the user interface.

- **Edge.** An edge is an object that is contained by a graph and connects nodes.

  An edge has a "head" and a "tail" as if it was directed, but also has a method isDirected() that says whether or not the edge should be treated as directed.

  An edge has a semantic object that is its semantic equivalent in the application and may have a visual object which is its syntactic representation in the user interface.

Application: Architecture for Interconnect System

4. **Port.** A Port is the interface of an Entity to any number of Relations.

   Normally, a Port is contained by an Entity, although a port may exist with no container.

   The role of a port is to aggregate a set of links to relations.

   Thus, for example, to represent a directed graph, entities can be created with two ports, one for incoming arcs and one for outgoing arcs.

5. **Relation.** A Relation links ports, and therefore the entities that contain them.

   To link a port to a relation, use the link() method in the Port class.

References

- Brooks C., Lee E.A., Liu X., Neuendorffer S., Zhao Y., and Zheng H., Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II), Department Electrical Engineering and Computer Sciences, Technical Report ECB/EECS-2008-28, University of California, Berkeley, CA, April, 2008.

- Chunithipaisan1 S., James P., Parker D., The Integration of Spatial Datasets for Network Analysis Operations, Department of Geomatics, University of Newcastle upon Tyne, Newcastle, UK, NE1 7 RU. DIS2004, pp. 123-132, August 2004.

- FutureEye 3.0: Computational Fluid Finite Elements, 2012.

- Lee E., Model-Driven Development – From Object-Oriented Design to Actor-Oriented Design, Presentation at Workshop for Software Engineering for Embedded Systems, From Requirements to Implementation, Chicago, September 24, 2003.

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