Abstract Classes and Interfaces

Mark A. Austin

University of Maryland

austin@umd.edu

ENCE 688P, Fall Semester 2020

October 12, 2020

Overview

1. Quick Review
2. Framework for Component-based Design
3. Abstract Classes
4. Working with Interfaces
5. Farm Worker Source Code
6. Five Applications
   - Two Factories making Widgets
   - Parsing and Evaluation of Functions with JEval
   - Using Interfaces in Spreadsheets
   - Horstmann's Simple Graph Editor
   - Architecture for Block Interconnect System

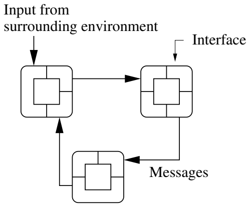
Quick Review

Quick Review: Objects and Classes

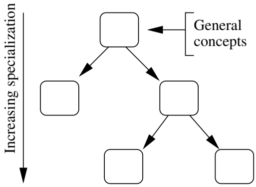
Motivating Ideas

- Simplify the way we view the real world,
- Provide mechanisms for assembly of complex systems.
- Provide mechanisms for handling systems that are subject to change.

Organizational and Efficiency Mechanisms



Network of Communicating Objects



Problem Domain Concepts organized into a Class Hierarchy.

Quick Review: Object-based Software

Basic Assumptions

- Everything is an object.
- New kinds of objects can be created by making a package containing other existing objects.
- Objects have relationships with other types of objects.
- Objects have type.
- Object communicate via message passing – all objects of the same type can receive and send the same kinds of messages.
- Objects can have executable behavior.
- Objects can be design to respond to occurrences and events.
- Systems will be created through a composition (assembly) of objects.

Quick Reiew: Objects and Classes

Working with Objects and Classes:

- Collections of objects share similar traits (e.g., data, structure, behavior).
- Collections of objects will form relationships with other collections of objects.

Definition of a Class

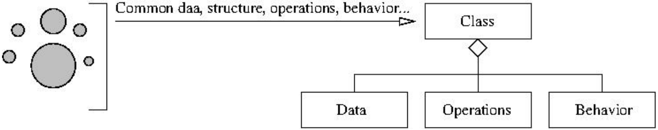
A class is a specification (or blueprint) of an object's structure and behavior.

Definition of an Object

An object is an instance of a class.

Quick Review: Objects and Classes

From Collections of Objects to Classes:



Common daa, structure, operations, behavior...

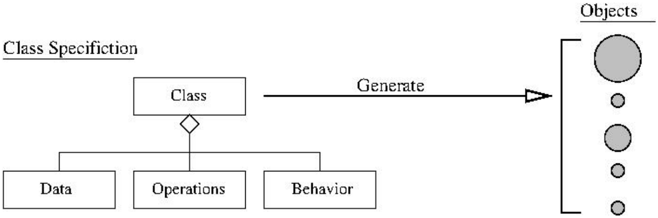
Class

Data

Operations

Behavior

Generation of Objects from Class Specifications:



Objects

Class Specifiction

Class

Generate

Data

Operations

Behavior

Quick Review: Objects and Classes

Key Design Tasks

- Identify objects and their attributes and functions,
- Establish relationships among the objects,
- Establish the interfaces for each object,
- Implement and test the individual objects,
- Assemble and test the system.

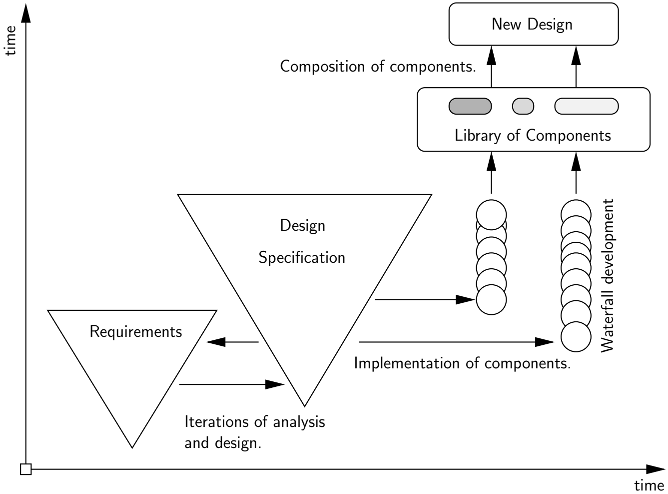
Implicit Assumptions → Connection to Data Mining

- Manual synthesis of the object model is realistic for systems that have a modest number of elements and relationships.
- As the dimensionality of the problem increases some form of automation will be needed to discover elements and relationships.

Framework for Component-based Design

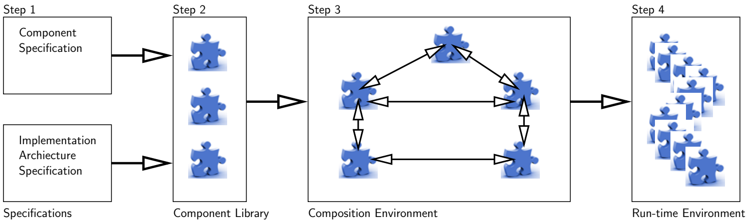
Framework for Component-based Design

Development for Reuse-Focused Design



Framework for Component-based Design

Simplified View of a Component Technology Supply Chain

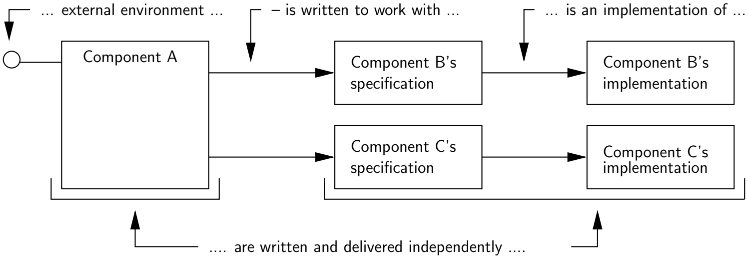


Implementation Requires

- Techniques for describing the overall system architecture.
- Definition of pieces in a way that facilitates assembly with other pieces (e.g., lego blocks).

Framework for Component-based Design

Simple Component-based Software System



... external environment ...

– is written to work with ...

... is an implementation of ...

Component A

Component B's specification

Component B's implementation

Component C's specification

Component C's 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.

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, ...

Interface-based Design

Interfaces are a specification for what an implementation should look like.

Benefits:

- Experience indicates that a focus on interfaces as a key design abstraction leads to designs with enhanced flexibility.
- Interface-based design procedures are particularly important for the design and managed evolution of systems-of-systems – e.g., cities.

Abstract Classes

Working with Abstract Classes

Abstract Classes

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.
- You cannot create an object from an abstract class – but they can be sub-classed.
- The subclasses will usually provide implementations for all of the abstract methods in its parent class.

Working with Abstract Classes

Example 1. Efficient Modeling of Shapes

A shape is a

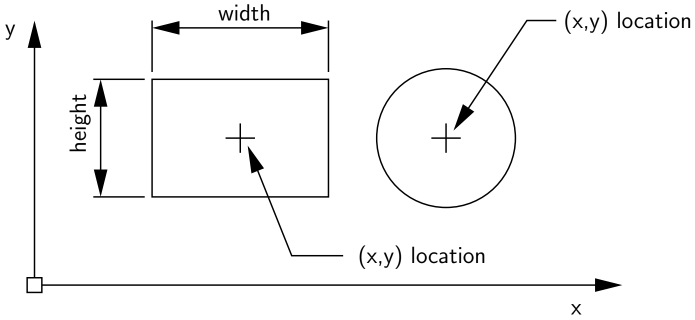
- High-level geometric concept that can be specialized into specific and well-known two-dimensional geometric entities.
- Examples: ovals, circles, rectangles, triangles, octogons, and so forth.

Capturing Shape Data

- There are sets of data values (e.g., vertex coordinates) and computable properties (e.g., area and perimeter) that are common to all shapes.

Working with Abstract Classes

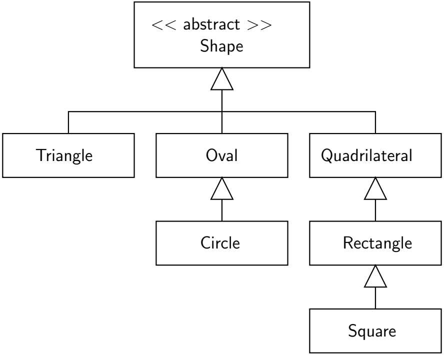
Capturing Shape Data



Computable properties: all shapes have an area, perimeter, an (x,y) centroid and a position or (x,y) location.

Working with Abstract Classes

Organizing Shapes into a Natural Hierarchy



<< abstract >>
Shape

Triangle

Oval

Quadrilateral

Circle

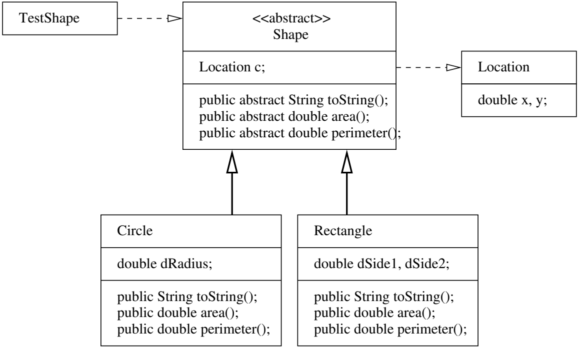
Rectangle

Square

Squares are a specific type of rectangle, which, in turn, are a specific type of quadralateral. Circles are a special type of oval.

Working with Abstract Classes

Class Diagram for TestShape Program



TestShape

<<abstract>>
Shape

Location c;

public abstract String toString();
public abstract double area();
public abstract double perimeter();

Location

double x, y;

Circle

double dRadius;

public String toString();
public double area();
public double perimeter();

Rectangle

double dSide1, dSide2;

public String toString();
public double area();
public double perimeter();

All extensions of Shape will need to provide 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, 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.

- The abstract shape class reduces the number of dependencies in the program architecture, making it ammenable to change – trivial matter to add Triangles to the class hierarchy.

Working with Abstract Classes

Walking Along an Array of Shapes

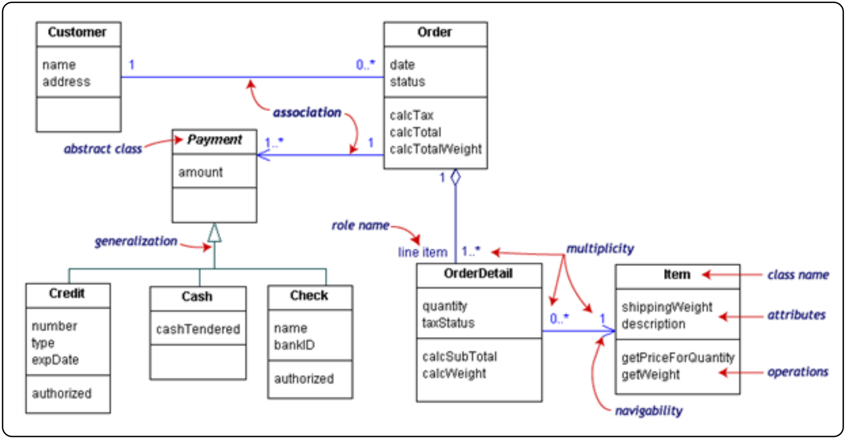
```
System.out.println("---------------------");
for (int ii = 1; ii <= s.length; ii = ii + 1) {
    System.out.println(  s[ii-1].toString() );
    System.out.println( "Perimeter = " + s[ii-1].perimeter() );
    System.out.println("---------------------");
}
```

Program Output:

```
---------------------
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
---------------------
```

Working with Abstract Classes

Example 2. Class Diagram for Operation of a Retail Catalog



Working with Abstract Classes

Points to Note:

- 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:

- Names of abstract classes, such as Payment, are in italics.
- Relationships between classes are the connecting links.

Working with Interfaces

Programming to an Interface

Motivation

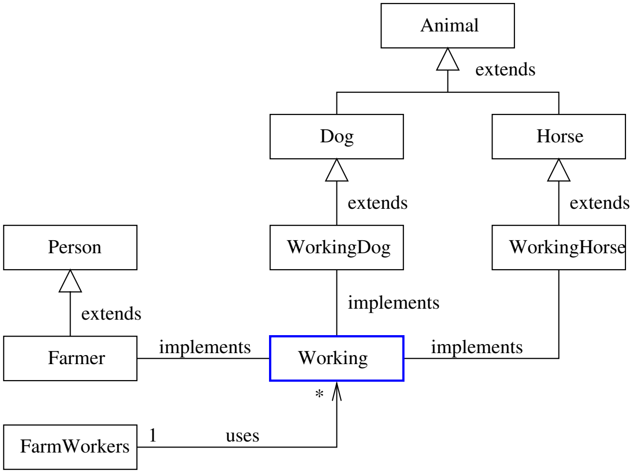
- Interfaces are the mechanism by which components describe what they do, but not how they do it.
- Interface abstractions are appropriate for collections of objects that provide common functionality, but are otherwise unrelated.

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.

Working with System Interfaces

Example 1. Software Interface for Farm Workers



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

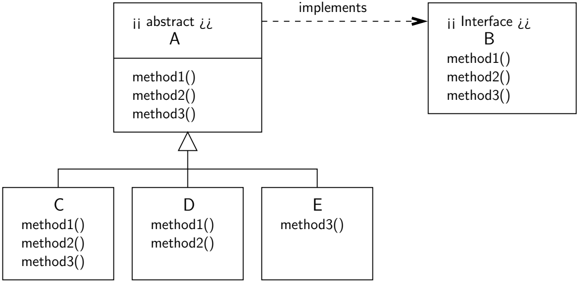
In Java, an interface represents what a class can do, but not how it will do it, which is the actual implementation.

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



¡¡ abstract ¿¿
A
method1()
method2()
method3()

implements

¡¡ Interface ¿¿
B
method1()
method2()
method3()

C
method1()
method2()
method3()

D
method1()
method2()

E
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();
=====================================  =====================
```

Farm Worker Source Code

Working with System Interfaces

Source Code: Animal.java

```java
public class Animal {
    String name;

    public Animal( String name ) { this.name = name; }
    public String toString()     { return this.name; }
}
```

Source Code: Dog.java

```java
public class Dog extends Animal {
    public Dog( String name ) { this.name = name; }

    public String toString(){
       return "*** In Dog: " + this.name;
    }
}
```

Source Code: Horse.java

```java
public class Horse extends Animal {
    public Horse( String name ) { this.name = name; }

    public String toString() {
       return "*** In Horse: " + this.name;
    }
}
```

Working with System Interfaces

Source Code: WorkingDog.java

```
public class WorkingDog extends Dog implements Working {
    public WorkingDog( String name ) {
       this.name = name;
    }

    public void hours () {
       System.out.println ( "*** Working dog hours -- working weekends!!" );
    }
}
```

Source Code: WorkingHorse.java

```
public class WorkingHorse extends Horse implements Working {
    public WorkingHorse( String name ) {
       this.name = name;
    }

    public void hours () {
       System.out.println ( "*** Working horse hours -- also working weekends!!" );
    }
}
```

Source Code: Working.java (Interface)

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

Working with System Interfaces

Source Code: Person.java

```
/*
 * ==================================================================
 * Person.java. Create person objects and compute their age...
 *
 * Written By: Mark Austin                              December 2006
 * ==================================================================
 */

import java.util.Calendar;
import java.util.Date;
import java.util.GregorianCalendar;

public class Person {
   protected String   sName;
   protected Date birthdate;

   // ===============================================================
   // Set/get name of a person
   // ===============================================================

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

   public String getName() {
      return sName;
   }
```

Working with System Interfaces

Source Code: Person.java (continued)

```

   // =============================================================
   // Compute age of a person ...
   // =============================================================

   public int getAge() {
      ... details removed ...
   }

   public void setBirthDate(Date aBirthDate) {
      this.birthdate = aBirthDate;
   }

   public void setBirthDate(int iYear, int iMonth, int iDay ) {
      Calendar cal = Calendar.getInstance();
      cal.set( iYear, iMonth, iDay );
      this.birthdate = cal.getTime();
   }

   public Date getBirthDate() {
      return birthdate;
   }

   // =========================================================
   // Create a String description of a persons cridentials
   // =========================================================
```

Working with System Interfaces

Source Code: Person.java (continued)

```

   public String toString() {
      String s = "Name: " + getName() + "\n";
            s += " Age: " + getAge() + "\n";
      return s;
   }
}
```

Source Code: Farmer.java

```
public class Farmer extends Person implements Working {
    public Farmer() {
       super();
    }

    public Farmer( String name ) {
       super();
       this.sName = name;
    }

    public String toString() {
       return "*** In Farmer: " + this.sName;
    }

    public void hours () {
       System.out.println ( "*** Working farmer -- working 7 days a week!!" );
    }
}
```

Working with System Interfaces

Source Code: FarmerWorkers.java (Test Program) ....

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

       // Create objects for farmers ....

       Working mac = new Farmer( "Old MacDonald" );
       System.out.println( mac.toString() );
       mac.hours();

       // Create objects for working farm animals ..

       Working max = new WorkingDog( "Max" );
       System.out.println( max.toString() );
       max.hours();

       Working silver = new WorkingHorse( "Silver" );
       System.out.println( silver.toString() );
       silver.hours();
   }
}
```

Working with System Interfaces

Test Program Output:

```
*** In Farmer: Old MacDonald
*** Working farmer -- working 7 days a week!!
*** In Dog: Max
*** Working dog hours -- working weekends!!
*** In Horse: Silver
*** Working horse hours -- also working weekends!!
```

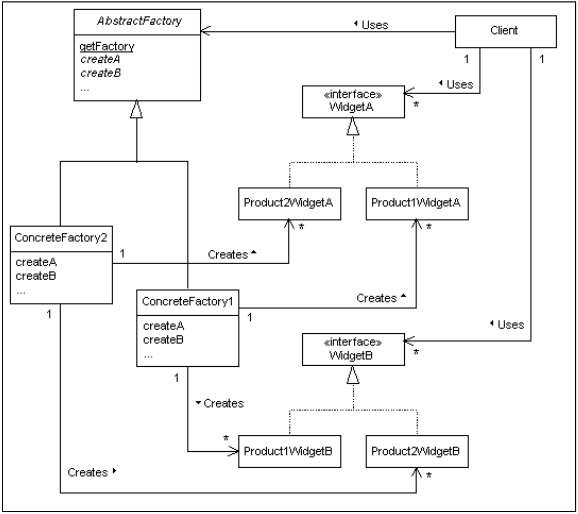
You might wonder:

Can I use this approach to call methods that are within a participating class (e.g., WorkingHorse), but not defined in the interface?

- No! You can only call methods defined in the interface.

Five Applications

Application 1. Two Factories making Widgets



Application 1. Two Factories making Widgets

Points to Note:

- The client works with an abstract model of a factory and two types of widgets, A and B, but only knows about their interfaces.
- The interfaces separate the client from details of how A and B are manufactured.
- Thus, a factory can change and the client will be completely unaware.

Application 2. Parsing/Evaluation of Functions with JEval

Purpose:

- JEval parses and evaluates dynamic and static expressions at run time.
- As such, it is a great solution for filtering streams of data at runtime.

Features:

- 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 2. 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 2. 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 2. 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`

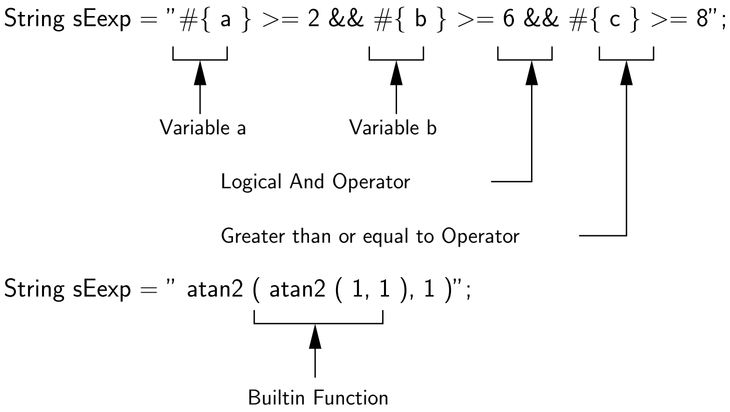
Application 2. Evaluation of Functions with JEval

Builtin Operator Functions:

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

Application 2. 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 2. Evaluation of Functions with JEval

Function Interface

```
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

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

Application 2. Evaluation of Functions with JEval

Operator Interface

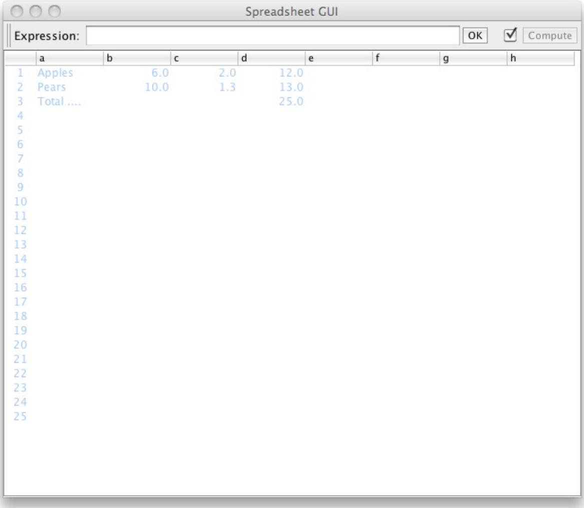
```
public interface Operator {
   // Evaluates two double operands.
   public abstract double evaluate(double  leftOperand,
                                   double rightOperand );

   // 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 3. Using Interfaces in Spreadsheets

Application 3: Graphical Interface



	a	b	c	d	e	f	g	h
1	Apples	6.0	2.0	12.0				
2	Pears	10.0	1.3	13.0				
3	Total			25.0				
4								
5								
6								
7								
8								
9								
10								
11								
12								
13								
14								
15								
16								
17								
18								
19								
20								
21								
22								
23								
24								
25								

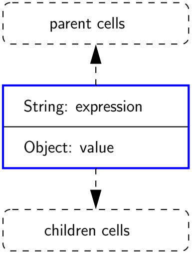
Application 3. Using Interfaces in Spreadsheets

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 3. Using Interfaces in Spreadsheets

Basic Cell Model



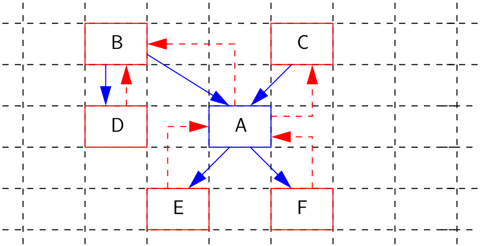
parent cells

String: expression

Object: value

children cells

Network of Dependencies among Cells



- The parents of Cell A are cells B and C; the children are cells E and F.
- No loops in the graph of dependency relationships.
- Topological sort → update cell values in one pass.

Application 3. Using Interfaces in Spreadsheets

Basic Spreadsheet Interface

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

   // 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 3. Using Interfaces in Spreadsheets

Extended Spreadsheet Interface

```
public interface IterableSpreadsheetInterface extends SpreadsheetInterf

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

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

   // Set/get the maximum change in value between successive loop itera

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

   // Recompute value of all cells ...

   public void recomputeWithIteration();
}
```

Application 3. Using Interfaces in Spreadsheets

Creating the Spreadsheet Model

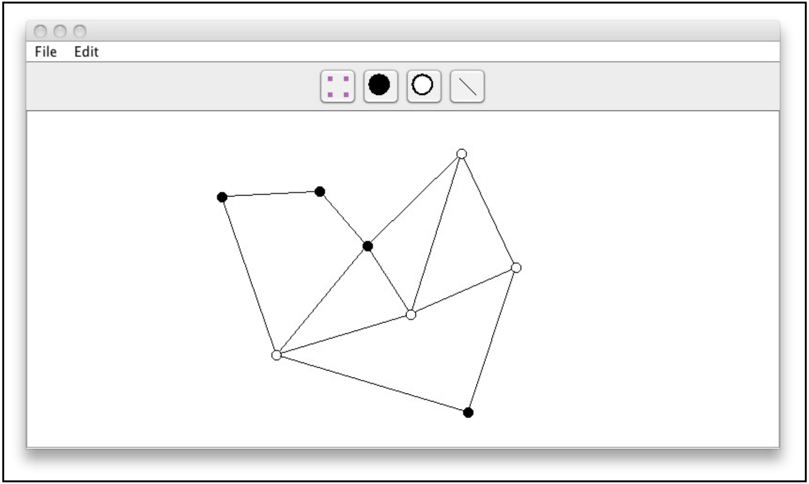
```
public class Spreadsheet implements SpreadsheetInterface {
   private int numRows, numColumns;     // no. of rows and cols
   private Map<String, Cell> cells;     // collection of all cells
   private String lastCellLocation;     // last cell accessed

   // 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 3. Using Interfaces in Spreadsheets

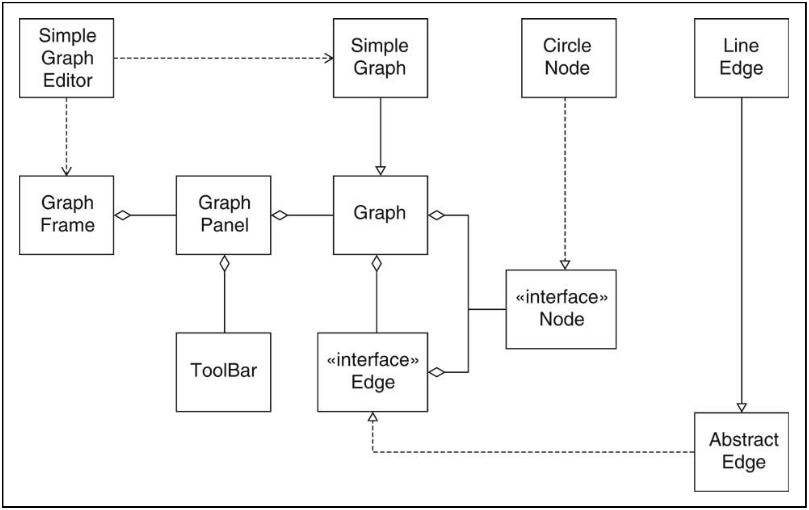
Creating a Spreadsheet Object

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

SpreadsheetInterface spreadsheet = new Spreadsheet(rows, columns);

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

Application 4. Horstmann's Simple Graph Editor



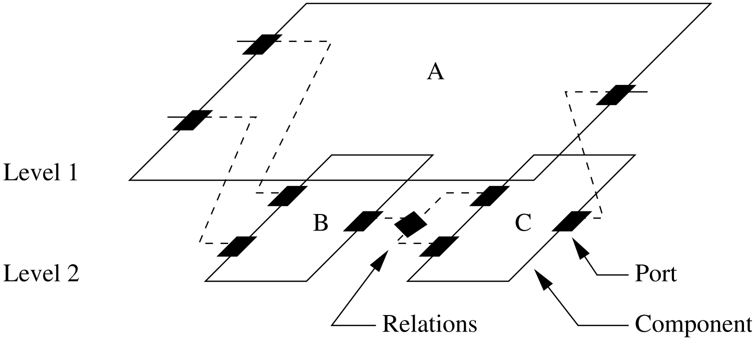
Application 4. Horstmann's Simple Graph Editor



Application 5. Architecture for Interconnect System

Problem Statement.

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



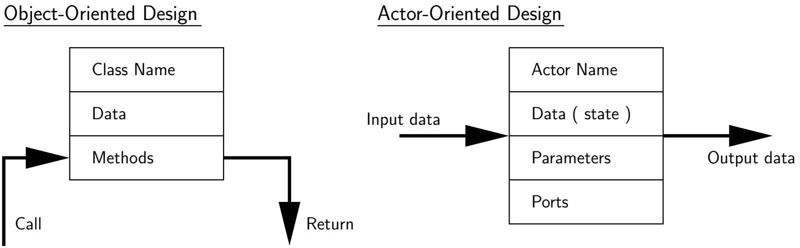
Application 5. Architecture for Interconnect System

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 5. Architecture for Interconnect System

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



Object-Oriented Modeling and Design

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

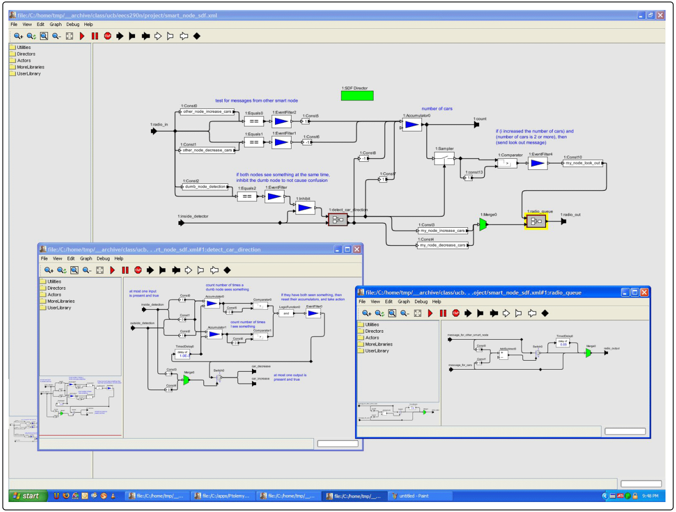
Application 5. Architecture for Interconnect System

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 5. Architecture for Interconnect System

Typical Ptolemy Application (see Brooks et al., 2008)



file:/C:/home/tmp/__archive/class/ucb/eecs290n/project/smart_node_sdf.xml

test for messages from other smart node

number of cars

if (i increased the number of cars) and
(number of cars is 2 or more), then
(send look out message)

if both nodes see something at the same time,
inhibit the dumb node to not cause confusion

file:/C:/home/tmp/__archive/class/ucb. . .rt_node_sdf.xml#1:detect_car_direction

at most one input
is present and true

count number of times a
dumb node sees something

If they have both seen something, then
reset their accumulators, and take action

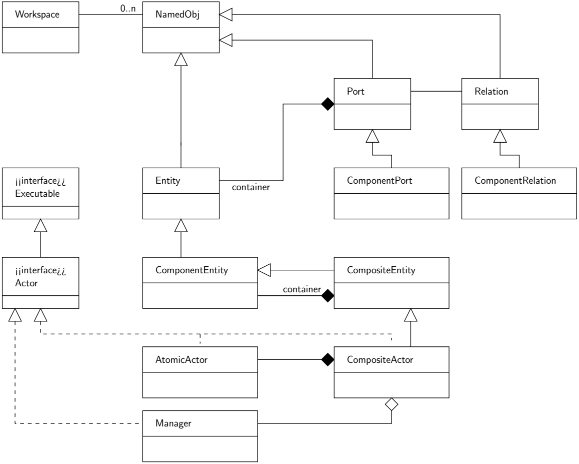
count number of times
I see something

at most one output is
present and true

file:/C:/home/tmp/__archive/class/ucb. . .oject/smart_node_sdf.xml#1:radio_queue

Application 5. Architecture for Interconnect System

Class diagram for modeling of system architectures in Ptolemy.



Application 5. 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.
- **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.

Application 5. Architecture for Interconnect System

- **Port.** A Port is the interface of an Entity to any number of Relations. 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.
- **Relation.** A Relation links ports, and therefore the entities that contain them.