California State University, East Bay
Department of Mathematics and Computer Science

CS 6310 ADVANCED SOFTWARE ENGINEERING

Introduction to the Unified Modeling Language and Design Patterns

Copyright 2008
Edward A. Billard
## Contents

I  Introduction to UML and Java .............................................................. 4

1 UML Notation ....................................................................................... 5

2 UML Examples ..................................................................................... 13

3 Exceptions ........................................................................................ 20

4 Java Database Connectivity (JDBC) ...................................................... 23

II  Data Structures in UML and Java ...................................................... 26

5 Stack (LIFO) ........................................................................................ 27

6 Queue (FIFO) ...................................................................................... 29

III  Patterns ............................................................................................ 34

7 Template/Factory Method Pattern: Specialization ................................. 35

8 Singleton Pattern: One Global Object .................................................. 36

9 Dynamic Starter Pattern: Runtime Decision ......................................... 38

10 Abstract Factory Pattern: Platform Independent GUI ............................. 40

11 Object Pool Pattern: Reusable JDBC Connections ............................... 42

12 Observer Pattern: Update Notification ................................................ 44

13 Snapshot Pattern: Persistent Objects ................................................... 46

14 Remote Proxy Pattern: Distributed Objects .......................................... 47

15 Composite Pattern: JUnit Testing ........................................................ 48

16 Composite Example: Document Parent to Child .................................... 57

17 Chain of Responsibility Pattern: Child to Parent ................................. 58

18 Builder Pattern: Hidden Construction ................................................ 59

IV  Case Study ......................................................................................... 60

19 Observable/Persistent Objects ............................................................. 61

20 Remote(RMI)/Persistent Objects ......................................................... 75
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>V Persistence Framework</td>
<td>81</td>
</tr>
<tr>
<td>21 3-Tier Architecture</td>
<td>82</td>
</tr>
<tr>
<td>22 POST Database</td>
<td>84</td>
</tr>
<tr>
<td>23 Virtual Proxies</td>
<td>85</td>
</tr>
<tr>
<td>24 POST Virtual Proxies</td>
<td>86</td>
</tr>
<tr>
<td>25 Brokers</td>
<td>87</td>
</tr>
<tr>
<td>26 Cache</td>
<td>89</td>
</tr>
<tr>
<td>27 Broker Server</td>
<td>93</td>
</tr>
<tr>
<td>28 Commit/Rollback</td>
<td>96</td>
</tr>
<tr>
<td>VI Tcl</td>
<td>97</td>
</tr>
<tr>
<td>29 Tcl Notes</td>
<td>97</td>
</tr>
<tr>
<td>30 Tcl Examples</td>
<td>103</td>
</tr>
<tr>
<td>31 Tcl Project</td>
<td>108</td>
</tr>
<tr>
<td>32 Tcl Tutorial</td>
<td>110</td>
</tr>
</tbody>
</table>
Part I
Introduction to UML and Java

- Unified Modeling Language (UML) integrated other modeling languages in mid 1990s
- UML allows software engineers to perform analysis and design in a consistent format
- Analysis tends to be classes and relationships between classes
- Design tends to be objects with attributes and methods
- Preliminary design consists of architectural features including inheritance and assembly
- Detailed design is one step away from programming, using collaboration/sequence diagrams
- UML allows for a systematic presentation of “design patterns”
- Design patterns are reusable software architectures that solve recurring problems
- No need to reinvent the wheel
- Example Problem: How to port application software to new platforms without modifications?
- Solution: The Abstract Factory Pattern
- This course introduces UML and Design Patterns, with implementations in Java
Class Definition:

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
</tr>
<tr>
<td>attribute2:type</td>
</tr>
<tr>
<td>attribute3:type=initial value</td>
</tr>
<tr>
<td>-privateAttribute</td>
</tr>
<tr>
<td>#protectedAttribute</td>
</tr>
<tr>
<td>+publicAttribute</td>
</tr>
<tr>
<td>classAttribute</td>
</tr>
</tbody>
</table>

| method1()                   |
| method2():type              |
| method3(parameter:type)      |
| -privateMethod()             |
| #protectedMethod()           |
| +publicMethod()              |
| classMethod()                |

class Class Name {
    int attribute2;
    int attribute3=10;
    private int privateAttribute;
    protected int protectedAttribute;
    public int publicAttribute;
    static int classAttribute;

    void method1() {}
    int method2() {return 0;}
    void method3(int parameter) {}
    private void privateMethod() {}
    protected void protectedMethod() {}
    public void publicMethod() {}
    static void classMethod() {}
}
Abstract Class Definition:

```java
abstract class AbstractClassName {
    abstract void abstractMethod();
}
```

Generalization/Specialization ("is a"):

```
class SuperClass {}
class SubClassX extends SuperClass {}
class SubClassY extends SuperClass {}
```

Realization ("realizes"):

```java
interface InterfaceName {
    public void operation();
}
```
Navigability ("has a") where the Association is "uses":

```
class Client {
    Server s;
    void clientMethod() {
        s.serverOperation();
    }
}
class Server {
    void serverOperation() {}
}
```

Aggregation (should be dark diamond):

```
class Part {}
class Whole {
    Part part1;
    Part part2;
    Part part3;
    Part part4;
}
```

- Aggregation creates and hides sub-parts
Static Structure Diagrams

Composition (is a clear diamond):

```
class Container {
    java.util.Vector v;
    void addChild(Object o) { v.addElement(o); }
}
```

- Composition is a container that receives objects from the outside

Multiplicity:

```
class Parent {
    java.util.Vector v1;
    java.util.Vector v2;
    Object o1;
    Object o2;
    Object o3;
}
```
package gui.framework;
class MyClass1 {}

package gui.servers;
import gui.framework;
class MyClass3 extends MyClass1 {}
Collaboration Diagram:

- firstMessage() in Class1, triggers 7 method calls
- classMethod() is a static call to Class2: Class2.classMethod()
- create() in UML is construction: “new”
- apply a method to an instance: id2.message1(99)
- 4 is an “if” statement: [some boolean condition]
- 5b is the “else” to the 5a “if”
- “for” loop: 6*: [i:=1..10]
- “while” loop (not shown): #*: [some boolean condition]
- 7: size() applied to collection of objects, not an individual object
class Class1 {
    private Class2 id2;
    private Class2 id3 = new Class2();
    private Class2 id4 = new Class2();
    private java.util.Vector v = new java.util.Vector();
    public Class1() {
        v.addElement(new Class2());
        v.addElement(new Class2());
        v.addElement(new Class2());
    }
    public void firstMessage() {
        boolean test = true;
        boolean test2 = true;
        Class2.classMethod(); // 1:
        id2 = new Class2(); // 2:
        id2.message1(99); // 3:
        if (test) id2.message2(); // 4:
        if (test2)
            id2.message3(); // 5a:
        else
            id3.message4(); // 5b:
        for (int i=1; i<=10; i++)
            id4.message5(); // 6:
        int size = v.size(); // 7:
    }
}

class Class2 {
    public static void classMethod() {}
    public void message1(int parameter){}
    public void message2(){}
    public void message3(){}
    public void message4(){}
    public void message5(){}
}

class Class3 {
    private Class1 id1 = new Class1();
    public Class3() {
        id1.firstMessage();
    }
}
Sequence Diagram:

```java
class Sequence1 {
    private Sequence2 id2 = new Sequence2();
    public void message1() { id2.message2(); }
}
class Sequence2 {
    public void message2() {
    }
}
class Sequence3 {
    private Sequence1 id1 = new Sequence1();
    public Sequence3() {
        id1.message1();
    }
}
```

- Collaboration Diagram uses numbers to illustrate triggering events
- Sequence Diagram uses vertical time line to illustrate triggering events
Collaboration Diagram:

- Given: class A has a private variable b of class B.

  ![Collaboration Diagram]

- x() triggers 1, which triggers 1.1, which triggers 1.1.1 and 1.1.2

```java
public class A {
    private B b = new B();
    public void x() {
        y();  // 1: is applied to self
    }
    private void y() {
        z();  // 1.1: is applied to self
    }
    private void z() {
        b.q(this);  // 1.1.1: is applied to b:B
        r();  // 1.1.2: is applied to self
    }
    private void r() {
    }
    public void s() {
    }
}

class B {
    public void q(A a) {
        a.s();  // 1.1.1.1: is applied backwards to a:A
    }
}
```
class A {
    public void x() {
        B b = new B(); // 1:
        C c = new C(); // 2:
        D d = new D(); // 3:
        b.y(this, c, d); // 4:
        c.z(this, d); // 5:
        q(); // 6:
    }
    public void q() {}  
}
class B {
    public void y(A a, C c, D d) {
        d.r(); // 4.1:
        c.z(a, d); // 4.2:
    }
}
class C {
    public void z(A a, D d) {
        d.r(); // 4.2.1: and 5.1:
        a.q(); // 4.2.2: and 5.2:
    }
}
class D {
    public void r() {}  
}
UML Examples

Static Structure Diagram:

```
A
+op1()

B
-c:C
-d:D
+B()
+op2()

C
-b:B
-d:D
+C(b:B)
+op3(d:D)
+op5()

D
-b:B
-c:C
+D(b:B)
+op4(c:C)
+op5()
```

Collaboration Diagram:

```
create()

:B
3.1.1:op1() 4.1.1:op1()
1:create(this)
3:op3(d)
4.1:op2()

:C
3.1:op5() 4.1:op5()

:D
2:create(this)
4:op4(c)
3.1.1:op2()
```

- What is the Java code?
Sequence Diagram:

```java
class A {
    private B b = new B();
    private C c = new C();
    public x() {
        b.y(c);
        b.r();
        c.s();
        t();
    }
    private t() {
    }
}

class B {
    public y(C c) {
        c.z(this);
    }
    public q() {
    }
}

class C {
    public z(B b) {
        b.q();
    }
}
```

- What is the Collaboration Diagram?
UML Example: Geometric Objects

Static Structure Diagram (Inheritance):

```
<table>
<thead>
<tr>
<th>Primitive</th>
<th>Conic</th>
</tr>
</thead>
<tbody>
<tr>
<td>+X:int=100</td>
<td>+a,b,c,d,e,f:int</td>
</tr>
<tr>
<td>+Y:int=100</td>
<td></td>
</tr>
<tr>
<td>-x:int</td>
<td>Conic(name,x,y,color,a,b,c,d,e,f)</td>
</tr>
<tr>
<td>-y:int</td>
<td></td>
</tr>
<tr>
<td>-name:String</td>
<td></td>
</tr>
<tr>
<td>#color:Color</td>
<td></td>
</tr>
<tr>
<td>Primitive(name,x,y,color)</td>
<td></td>
</tr>
<tr>
<td>setx(x)</td>
<td></td>
</tr>
<tr>
<td>sety(y)</td>
<td></td>
</tr>
<tr>
<td>getx():int</td>
<td></td>
</tr>
<tr>
<td>gety():int</td>
<td></td>
</tr>
<tr>
<td>getName():String</td>
<td></td>
</tr>
<tr>
<td>getColor():Color</td>
<td></td>
</tr>
<tr>
<td>chgAppearance()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+PI:double=3.14159</td>
</tr>
<tr>
<td>+num_circles:int=0</td>
</tr>
<tr>
<td>+r:int</td>
</tr>
<tr>
<td>Circle(name,r)</td>
</tr>
<tr>
<td>Circle(r)</td>
</tr>
<tr>
<td>circumference():double</td>
</tr>
<tr>
<td>area():double</td>
</tr>
<tr>
<td>bigger(c:Circle):Circle</td>
</tr>
<tr>
<td>bigger(a:Circle,b:Circle):Circle</td>
</tr>
<tr>
<td>chgAppearance()</td>
</tr>
</tbody>
</table>
```

- Application illustrates inheritance, visibility, class vs. instance attributes/methods
import java.awt.Color;  // package
public class Geo {
  public static void main(String args[]) {   // entry point
    Circle a = new Circle("a",10);        // object instance of class
    Circle b = new Circle("b",20);        // another object
    Color color;
    // access class variable
    System.out.println("Number of circles: "+Circle.num_circles);
    // access instance method
    System.out.println("Circ of a: "+a.circumference() + " Area of b: "+b.area());
    // access public instance variable
    System.out.println("Radius of a: "+a.r);
    // access instance method of grandparent
    System.out.println("Location of a: "+a.getx()+","+a.gety());
    // access instance method
    System.out.println("Biggest area: "+a.bigger(b).area());
    // access class method
    System.out.println("Biggest area: "+Circle.bigger(a,b).area());
    // access instance method of grandparent
    System.out.println("Circle with biggest area: "+a.bigger(b).getName());
    color = a.getColor();
    System.out.println("Color of circle a: "+color.toString());
    a.chgAppearance();   // make it darker
    color = a.getColor();
    System.out.println("Color of circle a: "+color.toString());
  }
}

abstract class Primitive {    // abstract=>cannot make an object instance
  public static final int X=100;  // class constant
  public static final int Y=100;
  private int x;                  // instance variable
  private int y;
  private String name;           // String is class
  protected Color color;         // let subclasses have access

  public Primitive(String name, int x, int y, Color color) { /* constructor
    this.name = name;           // refer to self
    this.x = x;
    this.y = y;
    this.color = color;
  }
  public void setx(int x) {this.x = x;}  // instance method
  public void sety(int y) { this.y = y; }
  public int getx() { return x; }
  public int gety() { return y; }
  public String getName() { return name; }
  public Color getColor() { return color; }
  public void chgAppearance() { color = color.brighter(); }
}
class Conic extends Primitive {  // inheritance

    private int a;  // ax^2 + bxy + cy^2 + dx + ey + f = 0
    private int b;
    private int c;
    private int d;
    private int e;
    private int f;

    public Conic(String name, int x, int y, Color color,
        int a, int b, int c, int d, int e, int f) {
        super(name,x,y,color);  // call constructor of parent
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
        this.e = e;
        this.f = f;
    }
}

class Circle extends Conic {  // inheritance
    public static final double PI=3.14159;  // class constant
    public static int num_circles=0;  // class variable
    public int r;  // instance variable

    public Circle(String name, int x, int y, Color color, int r) {
        super(name,x,y,color,1,0,1,0,0,-r*r);  // call constructor of parent
        num_circles++;
        this.r = r;
    }

    public Circle(String name, int r) {  // multiple constructors
        this(name,Primitive.X,Primitive.Y,Color.red,r);
    }

    public Circle(int r) { this("",r); }
    public double circumference() { return 2*PI*r; }
    public double area() { return PI*r*r; }

    public Circle bigger(Circle c) {  // instance method
        if (c.r > this.r) return c; else return this;  // return myself
    }

    public static Circle bigger(Circle a, Circle b) {  // class method
        if (a.r > b.r) return a; else return b;
    }

    public void chgAppearance() {  // overriding method
        color = color.darker();  // access protected variable
    }
}
Consider the following C-like code for handling error conditions:

```c
if (function1()) // boolean return type indicates success/failure
    if (function2())
        if (function3())
            print("all functions OK");
        else
            print("error function3");
    else
        print("error function2");
else
    print("error function1");
```

Or this code:

```c
function1(&error); // call-by-reference error code indicates success/failure
if (!error) {
    function2(&error);
    if (!error) {
        function3(&error);
        if (!error)
            print("all functions OK");
    }
}
if (error)
    print("error: %d", error);
```

Instead of cascading “if” statements, write code that assumes good behavior:

```c
function1();
function2();
function3();
function3();
// handle any possible errors
```

- That is the point of Exception handling in Java
- Note that printing may not be a suitable method for “handling” error events
- A function may not be able to handle an event, and then must pass it along
class MyException extends Exception {
    public MyException() { super(); }
    public MyException(String s) { super(s); }
}

class MyOtherException extends Exception {
    public MyOtherException() { super(); }
    public MyOtherException(String s) { super(s); }
}

class MySubException extends MyException {
    public MySubException() { super(); }
    public MySubException(String s) { super(s); }
}

public class Throwtest {
    public static void main(String argv[]) {
        int i;
        try { i = Integer.parseInt(argv[0]); }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Must specify an argument");
            return;
        }
        catch (NumberFormatException e) {
            System.out.println("Must specify an integer argument");
            return;
        }
        a(i);
    }
}
public static void a(int i) {
    try {
        b(i);
    } catch (MyException e) {
        if (e instanceof MySubException) {
            System.out.print("MySubException: ");
        } else {
            System.out.print("MyException: ");
        }
        System.out.println(e.getMessage());
    } finally {
        System.out.println("Handled at point 1");
    }
}

public static void b(int i) throws MyException {
    int result;
    try {
        System.out.print("i="+i+" ");
        result = c(i);
        System.out.println("c(i)="+result);
    } catch (MyOtherException e) {
        System.out.println("MyOtherException: "+e.getMessage());
        System.out.println("Handled at point 2");
    }
}

public static int c(int i) throws MyException, MyOtherException {
    switch (i) {
    case 0: // processing resumes at point 1 above
        throw new MyException("input too low");
    case 1: // processing resumes at point 1 above
        throw new MySubException("input still too low");
    case 99: // input resumes at point 2 above
        throw new MyOtherException("input too high");
    default: return i*i;
    }
}
Most applications need access to a database

Most databases are relational (SQL) tables of rows and columns

Java has a built-in technique (JDBC) to access SQL databases

Applications need 7 key lines of JDBC:

```java
import java.sql.*;
public class jdbc {
    public static void main(String args[]) {
        try {
            // 1: load specific code to talk to mysql
            Class.forName("org.gjt.mm.mysql.Driver");

            // 2: establish network connection to server
            Connection con = DriverManager.getConnection(
                    "jdbc:mysql://gold.mcs.csueastbay.edu:3306/SuppDB", // server name, port, database name
                    "4311", // user id
                    "4311"); // password

            // 3: environment for queries
            Statement stmt = con.createStatement();

            // 4: SQL query of a supplier table
            stmt.execute("select * from S where STATUS > 10");

            // 5: rs encapsulates rows which match the query
            ResultSet rs = stmt.executeQuery();

            // 6: iterate through the rows with a cursor
            //      next() returns false when cursor goes beyond results
            while (rs.next()) {
                // 7: extract columns from the row pointed to by the cursor
                System.out.println(rs.getString("S_N0") + " " + rs.getInt("STATUS"));
            }
        } catch (Exception e) {System.out.println(e.getMessage());}
    }
}
```
Java Database Connectivity (JDBC)

Uniform Resource Locator (URL):

SYNTAX: protocol://host:port/database

String url = "jdbc:mysql://gold.mcs.csueastbay.edu:3306/SuppDB"

CLIENT driver must match specific SERVER:

SYNTAX: Class.forName(driverName)

Class.forName("org.gjt.mm.mysql.Driver");
Class.forName("ORACLE DRIVER");

Make a CONNECTION:

SYNTAX: DriverManager.getConnection(url,user,pass)

Connection con = DriverManager.getConnection(url,"4311","4311");

Create a STATEMENT:

Statement stmt = con.createStatement();

Execute a QUERY:

ResultSet rs = stmt.executeQuery("select * from SP where STATUS >=20");

int i = stmt.executeUpdate("update SP set QTY=400 where S_NO = 'S1'");

if (stmt.execute("select * from SP where STATUS >=20"))
\ \ true means SQL was a SELECT
\ \ false means SQL was a UPDATE/INSERT/DELETE

Get the RESULTS:

ResultSet rs = stmt.executeQuery();

Get the METADATA (or DATA DICTIONARY):

ResultSetMetaData md = rs.getMetaData();
Get the NEXT (FIRST) record:

```java
while(rs.next())
```

Get the COLUMN COUNT

```java
int n = md.getColumnCount();
```

Get the COLUMN NAME or INDEX:

```java
String name = md.getColumnColumnName(i);
int i = rs.findColumn("QTY");
```

Get the COLUMN TYPE or WIDTH:

```java
String type = md.getColumnTypeName(i);
int size = md.getColumnDisplaySize(i);
```

Get the DATA by INDEX or NAME:

```java
String s = rs.getString(i);
String s = rs.getString("QTY");
int j = rs.getInt(i);
double d = rs.getDouble(i);
Timestamp t = rs.getTimestamp(i);
```

CLOSE:

```java
rs.close();
stmt.close();
con.close();
```

TRANSACTION PROCESSING:

```java
try {
  con.setAutoCommit(false);  // not supported by mysql
  stmt = con.createStatement();
  stmt.executeUpdate("update SP set QTY=400 where S_NO = 'S1'");
  stmt.close();
  stmt = con.createStatement();
  stmt.executeUpdate("update S set CITY='London' where S_NO = 'S1'");
  stmt.close();
  con.commit();
  con.close();
} catch (Exception e) {
  try {
    con.rollback();
  } catch (Exception e1) {}
}
```
Part II

Data Structures in UML and Java

- Data Structures provide an opportunity to explore some initial software problems
- Stacks and Queues can be designed in UML, implemented in Java
Static Structure Diagram:

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>-STACK_SIZE:int=100</td>
</tr>
<tr>
<td>-EMPTY_TOS:int=-1</td>
</tr>
<tr>
<td>-top:int=EMPTY_TOS</td>
</tr>
<tr>
<td>-stack_array:int[STACK_SIZE]</td>
</tr>
<tr>
<td>+main(args:String[])</td>
</tr>
<tr>
<td>+Stack()</td>
</tr>
<tr>
<td>+isEmpty():boolean</td>
</tr>
<tr>
<td>+isFull():boolean</td>
</tr>
<tr>
<td>+push(i:int)</td>
</tr>
<tr>
<td>+pop():int</td>
</tr>
</tbody>
</table>

Collaboration Diagram:

- Stack is fixed size array, which only holds integers
- Stack maintains Last-In First-Out (LIFO) by incrementing/decrementing top
class Stack {

    public static void main(String args[]) {
        Stack s = new Stack();
        s.push(5);
        s.push(1);
        s.push(7);
        while (!s.isEmpty())
            System.out.println(s.pop());
    }

    private static final int STACK_SIZE = 100;
    private static final int EMPTY_TOS = -1;
    private int top = EMPTY_TOS;
    int stack_array[] = new int[STACK_SIZE];

    public Stack () {} 

    public boolean isEmpty() {
        return(top == EMPTY_TOS);
    }

    public boolean isFull() {
        return(top == STACK_SIZE-1);
    }

    public void push(int i) {
        if (!isFull())
            stack_array[++top] = i;
    }

    public int pop() {
        if (isEmpty())
            return 0;
        else
            return(stack_array[top--]);
    }
}
6 Queue (FIFO)

Static Structure Diagram:

<table>
<thead>
<tr>
<th>Queue</th>
<th>Node</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>head:Node=nil</td>
<td>+element:Object</td>
<td>Object()</td>
</tr>
<tr>
<td>tail:Node=nil</td>
<td>+next:Node</td>
<td>toString():String</td>
</tr>
<tr>
<td>main(args:String[])</td>
<td>isEmpty():boolean</td>
<td></td>
</tr>
<tr>
<td>Queue()</td>
<td>enqueue():void</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dequeue():void</td>
<td></td>
</tr>
</tbody>
</table>

Collaboration Diagram:

- Queue is linked list of nodes, each contains element object and next reference
- Queue maintains First-In First-Out (FIFO) by enqueueing at tail, dequeueing at head
- If first enqueue(), adjust head also to point to the only Node
- If last dequeue(), adjust tail also to be null
Queue1: UML

Static Structure Diagram (Pattern: Delegation)

```
Queue1

main(args: String[])
Queue1()
isEmpty(): boolean
enqueue(obj:Object)
dequeue(): Object
```

```
Vector

addElement(obj:Object)
elementAt(i:int)
removeElementAt(i:int)
isEmpty(): boolean
```

```
Object

Object()
toString(): String
```

Collaboration Diagram:

```
main()

Queue1

1: q:=create()
3: enqueue(i)
4: obj:=dequeue()
2: i:=create(5)

<< new >>
Queue1
1.1: super()
3.1: addElement(obj)
4.1: elementAt(0)
4.2: removeElementAt(0)

<< new >>
Vector

<< new >>
Integer
```

- Queue1 "has a" Vector
- Instead of nodes, delegate most work to a Vector
- enqueue(): addElement() to the end of the Vector
- dequeue(): elementAt(0) is the head, removeElementAt(0) cuts it out of the Vector
import java.util.Vector;
public class Queue1 {

    public static void main(String args[]) {
        try {
            Queue1 q = new Queue1();
            q.enqueue(new Integer(5));
            q.enqueue(new Integer(1));
            q.enqueue(new Integer(7));
            while (!q.isEmpty())
                System.out.println(q.dequeue());
        } catch(Exception e) {System.out.println(e.getMessage());}
    }

    private Vector v;
    public Queue1() { v = new Vector(); }
    public boolean isEmpty() { return(v.isEmpty()); }
    public void enqueue(Object obj) {
        v.addElement(obj);
    }
    public Object dequeue() throws Exception {
        if (isEmpty())
            throw new Exception("Empty Queue");
        else {
            Object obj = v.elementAt(0);
            v.removeElementAt(0);
            return obj;
        }
    }
}

Queue2: UML

Static Structure Diagram (Inheritance):

```
Vector

addElement(obj:Object)
.elementAt(i:int)
removeElementAt(i:int)
isEmpty():boolean

Queue2

main(args:String[])
Queue2()
isEmpty():boolean
enqueue(obj:Object)
dequeue():Object

Object

* Object()
  * toString():String
```

Collaboration Diagram:

```
main()

Queue2

1: q:=create()
3: enqueue(i)
4: obj:=dequeue()
2: i:=create(5)

<< new >>
:Queue2

<< new >>
:Integer

1.1: super()
3.1: addElement(obj)
4.1: elementAt(0)
4.2: removeElementAt(0)
```

- Instead of “has a” Vector, Queue2 “is a” Vector
- Instead of v.addElement(), just addElement()
- Application would be allowed to inspect inside Queue2 via elementAt(3), say
import java.util.Vector;
public class Queue2 extends Vector {

    public static void main(String args[]) {
        try {
            Queue2 q = new Queue2();
            q.enqueue(new Integer(5));
            q.enqueue(new Integer(1));
            q.enqueue(new Integer(7));
            Integer i = (Integer)q.dequeue();
            Integer j = (Integer)q.dequeue();
            Integer k = (Integer)q.dequeue();
            String s1 = i.toString();
            String s2 = j.toString();
            String s3 = k.toString();
            System.out.println(s1 + " " + s2 + " " + s3);
        } catch(Exception e) {System.out.println(e.getMessage());}
    }

    public Queue2() { super(); }

    public void enqueue(Object obj) {
        addElement(obj);
    }

    public Object dequeue() throws Exception {
        if (isEmpty())
            throw new Exception("Empty Queue");
        else {
            Object obj = elementAt(0);
            removeElementAt(0);
            return obj;
        }
    }
}
Part III
Patterns

- UML allows for a systematic presentation of “design patterns”
- Design patterns are reusable software architectures that solve recurring problems
- No need to reinvent the wheel
- Example Problem: How to port application software to new platforms without modifications?
- Solution: The Abstract Factory Pattern
public abstract class Template {
    public void normalOperation() {
        helperMethod();
        templateMethod(); // need some help, but it is too specialized
    }
    private void helperMethod() {
        // some generalized code inherited by all sub-classes
    }
    public abstract void templateMethod(); // cannot code here, need specialization
}

public class Concrete1 extends Template {
    public void templateMethod() {
        // some specialized code for just this sub-class
    }
}

public class Concrete2 extends Template {
    public void templateMethod() {
        // some specialized code for just this sub-class
    }
}

• Put GENERALIZED code high up in the hierarchy (re-used by sub-classes)
• But if SPECIALIZED then need to have sub-classes code it
• That's ALL that an abstract method does
• Formal Name: Template Method
• Factory Method: same as Template, but the method returns a new specialized object

public class Concrete1 extends Template {
    public myObject factoryMethod() {
        return new myObject1(); // where myObject1 IS A myObject
    }
}
Problem: How to create just one instance of a globally-accessible object?

Solution: Singleton Pattern

Outline: Instead of making all attributes and methods static, make just one static attribute (instance), whose type is the globally-accessible object, along with a static method, which returns this instance.

Original Code:

```java
public class AudioClipManager {
    private AudioClip prevClip;
    public AudioClipManager() {}
    public play(:AudioClip) {}
    public loop(:AudioClip) {}
    public stop() {}
}

AudioClipManager acm = new AudioClipManager();
acm.play(audioClip);
acm.loop(audioClip);
acm.stop();
```

Awkward conversion to all statics:

```java
public class AudioClipManager {
    private static AudioClip prevClip;
    // no constructor
    public static play(:AudioClip) {}
    public static loop(:AudioClip) {}
    public static stop() {}
}

AudioClipManager.play(audioClip);
AudioClipManager.loop(audioClip);
AudioClipManager.stop();
```
public class AudioClipManager {
    // singleton constructs itself when class loader detects first reference
    private static AudioClipManager instance = new AudioClipManager();

    // normal instance data
    private AudioClip prevClip;

    // private constructor since singleton constructs itself
    // do not want a default public constructor from the compiler
    private AudioClipManager() {}

    public static AudioClipManager getInstance() {
        return instance;
    }

    // normal instance methods
    public play(AudioClip) {}
    public loop(AudioClip) {}
    public stop() {}
}

// one static call to get the instance
// first such call triggers class loading and construction of singleton
AudioClipManager acm = AudioClipManager.getInstance();

// the rest are just normal instance calls
acm.play(audioClip);
acm.loop(audioClip);
acm.stop();
9 Dynamic Starter Pattern: Runtime Decision

- **Problem:** How to decide at runtime which program to execute (e.g. user types in string name)?

- **Solution:** Dynamic Starter Pattern

- **Outline:** The String name of a class, with a start() method, is converted to a Class, converted to an Object, and then the start() method is called.

- UML is shown for #1 (which means ConcreteProgram1), but should be able to work for any number of ConcretePrograms

- Main() call is just used for testing purposes

- Normally, a Java client would decide at runtime which ConcreteProgram to run

- Net effect is that the start method of any ConcreteProgram can be invoked

- Program is tested with: java AbstractDynamicProgram ConcreteProgram1

Hard-coded program:

```java
class ConcreteProgram1 {
    public void start() {
        // beginning of code
    }
}

(new ConcreteProgram1()).start();
```

Awkward solution (not easily extended to other classes):

```java
class ConcreteProgram1 {
    public void start() {
    }
}
class ConcreteProgram2 {
    public void start() {
    }
}

switch (code) {
    case 1: (new ConcreteProgram1()).start();
        break;
    case 2: (new ConcreteProgram2()).start();
        break;
}
```
Dynamic Starter Pattern: Runtime Decision

- `Main()` provided `programName` to run
- `Class.forName()` returns a `Class`
- `newInstance()` returns an `Object` that can be cast as an `AbstractDynamicProgram`
- All `AbstractDynamicPrograms` have a `start()` method, similar to a `main()`
- `start` is italicized
- Classes with abstract methods must be abstract, method must be overloaded
- Abstract class without any abstract methods still forces subclasses
**Problem:** How to port application software to new platforms without modifications?

**Solution:** Abstract Factory Pattern

**Outline:** Each platform requires specific code to achieve implementation. Hide the construction of the specific objects in a Factory per platform. The application calls the Factory to do the construction rather than explicitly calling “new”.

Awkward solution is to keep a specialized source per platform:

```java
class GUI_SUN {
}
class GUI_MS {
}
class GUI_APPLE {
}
```

Another awkward solution is compile-time C language “defines” to do conditional inclusion:

```c
#define SUN 1
#define MS 2
#define APPLE 3
#define PLATFORM SUN

#if PLATFORM == SUN
   // put SUN code here
#elseif PLATFORM == MS
   // put MS code here
#elseif PLATFORM == APPLE
   // put APPLE code here
#endif
```

Abstract Factory solution:

- Code number (from environment file) determines platform
- Application (e.g. GUI) calls AbstractFactory.getFactory(code) to get a Factory
- Specialized sub-class Factory, implements the construction of specialized GUI objects
- Instead of calling “new”, application asks Factory to construct object
- Application deals with Menu and Button, not SunMenu, MSMenu, SunButton, MSButton
- Application deals with AbstractFactory, not Sun Factory, MSFactory
### Abstract Factory Pattern: Platform Independent GUI

<table>
<thead>
<tr>
<th>AbstractFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>sun: SunFactory</td>
</tr>
<tr>
<td>ms: MSFactory</td>
</tr>
<tr>
<td>getFactory(code:int):</td>
</tr>
<tr>
<td>AbstractFactory</td>
</tr>
<tr>
<td>createMenu(): Menu</td>
</tr>
<tr>
<td>createButton(): Button</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>createMenu(): Menu</td>
</tr>
<tr>
<td>createButton(): Button</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SunFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>createMenu(): Menu</td>
</tr>
<tr>
<td>createButton(): Button</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>createMenu(): Menu</td>
</tr>
<tr>
<td>createButton(): Button</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GUI</th>
</tr>
</thead>
<tbody>
<tr>
<td>main(args: String[])</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>select()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SunMenu</th>
</tr>
</thead>
<tbody>
<tr>
<td>select()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSMenu</th>
</tr>
</thead>
<tbody>
<tr>
<td>select()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>push()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SunButton</th>
</tr>
</thead>
<tbody>
<tr>
<td>push()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MSButton</th>
</tr>
</thead>
<tbody>
<tr>
<td>push()</td>
</tr>
</tbody>
</table>

1: code = parseInt(args[0])
2: f = getFactory(code)
3: m = createMenu()
4: b = createButton()
5: select()
6: push()

2.1: case 1: return sun
3.1: create()
4.1: create()
5.1: println("SMS")
6.1: println("SBP")

- Buttons and Menus are just stubs: push() just says “Sun Button Push”
- f:SunFactory because the code is #1; but in GUI it is f:AbstractFactory
• **Problem:** How to construct time-expensive objects at boot time, and reuse objects during runtime?

• **Solution:** Object Pool Pattern

• **Outline:** Instead of dynamically constructing time-expensive objects (e.g., network/database connections), preallocate and reuse the objects as necessary. Application acquires/releases an object from the pool instead of constructing the object each time it is needed.

• Pool is a collection of objects, one as good as another, that can be used, returned, and then reused

• Example: Pool of JDBC connections, each expensive to build, constructed at boot time

• Construct \( n \) connections to a database, store in a Vector, place in a Hashtable keyed on database name

• **ConnectionPool** is a Singleton, globally accessible

• **Main()** acquires a Connection, which causes a lookup in the Hashtable, and a dequeue from the Vector

• Using the Connection, main forms and executes an SQL query

• **Main()** releases Connection, which causes a lookup in Hashtable, and enqueue onto Vector
Object Pool Pattern: Reusable Objects

- ConnectionPool is a Singleton.
- Pool has a Hashtable, indexed by dbName, which returns a Vector.
- Each Vector contains n reusable JDBC Connection objects pre-connected to dbName.
• **Problem:** How to maintain consistency between multiple observers of an observable object’s state information?

• **Solution:** Observer Pattern

• **Outline:** Consider an Observable Supplier object with Number, Name, Status, City. Two Observer Frames both display the information about this particular Supplier:

| Supplier #: S1
| Name: Smith
| Status: 20
| City: London
| SUBMIT |
| Supplier #: S1
| Name: Smith
| Status: 20
| City: London
| SUBMIT |

• A user modifies two fields in the first Frame, and pushes SUBMIT

• The goal is to have the other Frame(s) automatically refresh their own displays:

| Supplier #: S1
| Name: Smith
| Status: 30
| City: Paris
| SUBMIT |
| Supplier #: S1
| Name: Smith
| Status: 30
| City: Paris
| SUBMIT |
- Supplier inherits from Java’s Observable class
- ObsFrame implements Java’s Observer interface
- Many ObsFrames observe the one Supplier
- Each ObsFrame registers itself with the Supplier via addObserver(this)
- addObserver() puts the Observer into the Vector of all Observers
- If a ObsFrame does a SUBMIT, then it mutates the Supplier
- Supplier calls notifyObservers()
- notifyObservers() iterates through the Vector, and calls each Observers’s update()
- update() is the automatic update (callback) which allows the ObsFrame to refresh the display
- Refresh can be performed by using the Supplier’s accessor methods
- SEE CASE STUDY
Problem: How to make an object persistent on the disk?

Solution: Snapshot Pattern

Outline: An in-memory object “lives” only as long as the application program is running. But an object might contain important data that needs to be retained essentially forever, in particular, objects which represent information in a database.

Alternative 1: Use Java’s serialization mechanism to implement Snapshot Pattern

Alternative 2: Use an SQL Relational database to implement Snapshot Pattern

For example, a relational table has a row per object:

```sql
mysql> select * from S;
+-----------+-------+-------+---------+-------+
| S_NO      | S_NAME| STATUS| CITY    |       |
|-----------+-------+-------+---------+-------|
| S1        | Smith | 20    | London  | 30    |
| S2        | Jones | 10    | Paris   |       |
| S3        | Blake | 30    | Paris   |       |
| S4        | Clark | 20    | London  |       |
| S5        | Adams | 30    | Athens  |       |
+-----------+-------+-------+---------+-------+

mysql> ```

When “S1” is constructed by Application, read the associated fields from the row

When “S1” mutates, modify the row to the new values

SEE CASE STUDY
• **Problem:** How to access a distributed object on the network as if the object was on the same computer?

• **Solution:** Remote Proxy Pattern

• **Outline:** The Proxy Pattern provides for external service by a proxy, or substitute, object. The Remote Proxy Pattern uses a Proxy with the same look-and-feel as a RealSubject located somewhere on the network. Both the Proxy and the RealSubject implement the same interface so as to have the same look-and-feel. A call to op1() on the Proxy results in a call to op1() across the network to the RealSubject.

![Remote Proxy Pattern Diagram]

- Java’s Remote Method Invocation (RMI) is the Remote Proxy Pattern
- The Proxy is called a “stub”
- The RealSubject has a “skeleton” that talks across the network to the stub
- SEE CASE STUDY
Life cycle of software engineering: requirements, analysis, design, implementation, test

- Testing steps: unit, integration, validation, system, regression
- Unit test: provide test case inputs to individual object, evaluate results for correctness
- White box unit test: examine paths in sources to develop test cases
- Black box unit test: examine specification of object to develop test cases
- Test harness: outer program to provide structure to hold the object to be tested
- JUnit: Java test harness, implemented with Composite Pattern, to execute a test suite of test cases
- JUnit is mostly focused on black box test
- Methods assertEquals(), assertTrue() strategically placed to evaluate results for correctness
- Assertions mainly test pre-conditions and post-conditions to method calls
- Exceptions that arise from assertions are called “failures”, and anticipated by the assertion
- Other exceptions that are not anticipated (e.g. division by zero) are called “errors”
- Test suite of test cases can be used to ensure quality when code changes are made later (i.e. regression)
- When a bug is reported, a JUnit test case can be developed to mimic the problem
- Developer uses the test case, fixes the problem, and tries to get a clean run without a failure
- Instead of implementation and then test case development: test-driven development
- Write JUnit test cases first, using only stubs of designed objects, most will generate failures
- Now start coding actual objects; finished coding when failures are eliminated
**Problem:** How to build a hierarchy of objects such that a common method among all of the objects can be invoked?

**Solution:** Composite Pattern

**Outline:** For example, the JUnit test harness requires that each test case of a test suite is “run”. Establish a Test interface with a run() method, implemented by individual TestCases as well as TestSuite collections.

Build a hierarchical tree structure where interior nodes are TestSuites, and leaf nodes are TestCases. Each TestSuite has a Vector of children, which may be either another TestSuite or an individual TestCase. When the run() method is invoked at the root (or any node) of the tree, then the run() method of all the children is invoked. This propagates to the leaves of the tree.

All run() methods share a parameter, TestResult, which is a log of any failure or error exceptions. At the end of the run, the log is printed to display the results.

In the example below, MyCalendarTest and VectorTest are TestCases, which are examined in more detail later.
Composite Pattern: Hierarchy

- TestSuite interior nodes have Vectors of many children, either TestSuites or TestCases
- Invocation of the run() method at the root propagates to all the run() methods at the leaves
JUnit: Framework Java Code

// Assert contains the methods used to test data for pre- and post-conditions
// An assertion which is not met causes an exception to be thrown
// Various signatures are provided to examine different types of data
package junit.framework;

public class Assert {
    public static void assertTrue(String s, boolean condition) {
        if (!condition)
            throw new AssertionError(s + "":" + condition);
    }
    public static void assertEquals(int expected, int actual) {
        if (expected != actual)
            throw new AssertionError(expected + "<>" + actual);
    }
    public static void assertEquals(String expected, String actual) {
        if (!expected.equals(actual))
            throw new AssertionError(expected + "<>" + actual);
    }
}

// AssertionError stores the error message for the exception
public class AssertionError extends Error {
    public AssertionError() { super(); }
    public AssertionError(String s) { super(s); }
}

// All TestCases and TestSuites implement Test, hence a run() method, containing a log
public interface Test {
    public void run(ResultSet result);
}

// TestSuites contain Vectors of all their children, each an implementation of Test
// Children are added via addTest()
// Invocation of the run() method triggers invocation of the run() methods of all children
public class TestSuite implements Test {
    private Vector fTests = new Vector();
    public void run(ResultSet result) {
        for (Enumeration e = fTests.elements(); e.hasMoreElements();)
            (Test)e.nextElement().run(result);
    }
    public void addTest(Test test) {
        fTests.addElement(test);
    }
}
JUnit: Framework Java Code

// TestFailure encapsulates the Test which caused a Throwable (exception)
// Using reflection, toString() formats for display by the log
public class TestFailure {
    private Test fFailedTest;
    private Throwable fThrownException;
    public TestFailure(Test test, Throwable t) {
        fFailedTest = test;
        fThrownException = t;
    }
    public Test failedTest() { return fFailedTest; }
    public Throwable thrownException() { return fThrownException; }
    public String toString() {
        TestCase test = (TestCase)fFailedTest;
        String className = test.getClass().getName();
        return className + "." + test.getName() + ": " + fThrownException.getMessage();
    }
}

// TestResult is a log of all errors and failures
// print() iterates through both Vectors and the toString() above yields the display
public class TestResult {
    protected Vector fErrors = new Vector();
    protected Vector fFailures = new Vector();
    public synchronized void addError(Test test, Throwable t) {
        fErrors.addElement(new TestFailure(test, t));
    }
    public synchronized void addFailure(Test test, Throwable t) {
        fFailures.addElement(new TestFailure(test, t));
    }
    public synchronized Enumeration errors() {
        return fErrors.elements();
    }
    public synchronized Enumeration failures() {
        return fFailures.elements();
    }
    public synchronized void print() {
        System.out.println("Errors:");
        for (int i=0; i<errors.size(); i++) {
            TestFailure testFailure = (TestFailure)errors.elementAt(i);
            System.out.println(testFailure);
        }
        System.out.println("Failures:");
        for (int i=0; i<failures.size(); i++) {
            TestFailure testFailure = (TestFailure)failures.elementAt(i);
            System.out.println(testFailure);
        }
    }
}
// A test case extends TestCase, hence inherits assertions and must have a run() method
// The test case is run and exceptions caught
// The exceptions (anticipated AssertionError or other unanticipated Errors) are logged
// The method String name (with assert code), is converted to a Method object
// Then invoke() is called on the Method object
public abstract class TestCase extends Assert implements Test {
    private final String fName;
    public TestCase(String name) { fName = name; }
    public String getName() { return fName; }
    public void run(TestResult result) {
        setUp();
        try {
            runTest();
        } catch (AssertionFailedError e) {
            result.addFailure(this, e);
        } catch (Throwable e) {
            result.addError(this, e);
        }
        finally {
            tearDown();
        }
    }
    public TestResult run() {
        TestResult result = new TestResult();
        run(result);
        return result;
    }
    public void runTest() throws Throwable {
        Method runMethod = null;
        try {
            runMethod = getClass().getMethod(fName, null);
        } catch (NoSuchMethodException e) {
            assertTrue("Method "+fName+" not found",false);
        }
        try {
            runMethod.invoke(this, null);
        } catch (InvocationTargetException e) {
            e.fillInStackTrace();
            throw e.getTargetException();
        }
        protected void setUp() {} 
        protected void tearDown() {}
    }
}
• Composite Pattern is used to integrate VectorTest, MyCalendarTest into a TestSuite

```java
import junit.framework.*;
public class AllTests {
    public static Test suite() {
        TestSuite A = new TestSuite();
        TestSuite B = new TestSuite();
        TestSuite suite = new TestSuite();
        suite.addTest(A);
        suite.addTest(B);

        // testSize(), testElementAt() are test methods on a Vector
        A.addTest(new VectorTest("testSize"));
        A.addTest(new VectorTest("testElementAt"));

        // testGetName() is the test method on a calendar
        B.addTest(new MyCalendarTest("testGetName"));
        return suite;
    }
    public static void main(String args[]) {
        Test allTests = suite();
        TestResult result = new TestResult();

        // one log is passed to all invocations of the run() method
        allTests.run(result);

        // after run, display log of exceptions
        result.print();
    }
}
```
// Simple calendar conversion of integer month to String month
public class MyCalendar {
    public static String getName(int month) {
        switch (month) {
            case 1: return "January";
            case 2: return "February";
            case 3: return "March";
            case 4: return "April";
            case 5: return "May";
            case 6: return "June";
            case 7: return "July";
            case 8: return "August";
            case 9: return "September"; // THIS IS NOT SPELLED CORRECTLY
            case 10: return "October";
            case 11: return "November";
            case 12: return "December";
            default: return "UNKNOWN";
        }
    }
}

// TestCase for MyCalendar using one method
import junit.framework.*;
public class MyCalendarTest extends TestCase {
    public MyCalendarTest(String s) {
        super(s);
    }
    public void testGetName() {

        // A guess is that 9 is not converted correctly to September
        // Local variable name is not required
        // The assertion will generate a failure exception to the log

        String name = MyCalendar.getName(9);
        assertEquals(name,"September");
    }
}
import junit.framework.*;
import java.util.Vector;
public class VectorTest extends TestCase {
    private Vector v;
    public VectorTest(String s) { super(s); }

    // All TestCases have an optional setUp() before each invocation of individual tests
    protected void setUp() {
        v = new Vector();

        // a new Vector should be empty
        assertTrue("isEmpty", v.isEmpty());

        // after adding an element, the Vector should NOT be empty
        v.addElement(new Integer(1));
        assertTrue("!isEmpty", !v.isEmpty());
        v.addElement(new Integer(2));
        v.addElement(new Integer(3));
    }

    // Each of these method's name was attached to the test case via AllTests.java
    // This can be cumbersome and JUnit can automatically identify test methods
    public void testSize() {
        int size = v.size();
        for (int i=0; i<100; i++)
            v.addElement(new Integer(i));

        // the size should have grown by 100 elements
        assertEquals(v.size(), size+100);
    }
    public void testElementAt() {

        // the original value added should be 1
        Integer i = (Integer)v.elementAt(0);
        assertEquals(i.intValue(), 1);
    }
}
16 Composite Example: Document Parent to Child

DocElement
getCharLen(): int

CompDocElement
addChild(child: DocElement)

Document 1 2

Page 1

Char 2

Diagram showing the relationship between Document, Page, and Char with methods and properties.
• IS-A Hierarchy: Building is a SecurityZone, Floor is a ...
• HAS-A Hierarchy: Building has many Floors, Floor has many Rooms, Room has many Sensors
• but Building does not maintain a list of children, as does Composite Pattern
• child keeps one pointer to parent
• control flows upwards from child to parent
• control flows downwards in Composite Pattern
• leaf of hierarchy (sensor) generates a notify of a problem
• each node checks to see if it can handle the problem
• if it cannot handle the problem, then notify the parent
• where is notify programmed?
• where is handle programmed?
• Document is an example of a Composite Pattern
• Document has many Pages, has many Lines, has many Chars
• getCharLen accumulates length over the Composite
• Builder hides the details of constructing complex objects, such as a Composite
• there can be many different types of Builder, based on code, but each has same part methods
• Director knows these part steps, and calls them in correct sequence
• where else could we put this code?
• entire Document can be retrieved by the Client at the end
Part IV

Case Study

- Goal: Incorporate the Observer, Persistent (Snapshot), Remote Patterns

- Ultimate Object: Many Observers can be notified upon updates, the object lives forever on the disk, and the object can be distributed somewhere else on the network.

- First, consider Observable and Persistent Objects

- Persistence achieved with SQL database or Java’s Serializable interface

- Second, consider Persistent and Remote Objects

- Remote Objects achieved with Java’s Remote Method Invocation (RMI)
• Multiple Observer Frames watch the Observable Supplier

• Each Frame registers with the Supplier using a call to addObserver(this)

• Supplier does addElement(observer) to the Vector in Observable

• User presses SUBMIT on a Frame, which mutates() the Supplier

• Supplier modifies the disk using SQL or Java’s Serializable interface

• Supplier calls inherited notifyObservers() 

• notifyObservers() iterates through Vector, casts, calls update() on Frame

• Each Frame has an update(), which calls the Supplier’s accessor methods

• Frame refreshes the screen
Observable/Persistent Classes: Responsibilities

- The following diagrams provide detail about the Class Responsibilities:

- Observable: maintain Vector of registered Observers, notifyObservers() upon mutation

- Observer: interface to enforce ObsFrame to have update() method

- Obsmain: create 2 Suppliers, and $n$ ObsFrames per to watch

- DataBase: perform first 5 (out of 7) steps of JDBC to query database

- ObsFrame: register as Observer, display 4 fields, button causes mutate(), update() causes display-Supplier()

- Supplier: maintain 4 fields, “is a” Observable, “has a” DataBase, upon construction loadSupplier() from DataBase, upon mutation storeSupplier() to the DataBase and notifyObservers()
private static final String driverName = "org.gjt.mm.mysql.Driver"

private String url = "jdbc:mysql://gold.mcs.csueastbay.edu:3306/"

- Database is constructed by the Supplier when it is constructed
- Constructor for Database initializes private variables
- Base URL indicates where the server is located and the port it uses
- URL needs to have the database name appended to the base URL
- Note the location of the drivers
- Supplier calls `execute()` to `loadSupplier()` from DB, or `storeSupplier()` to DB

- First 5 lines of JDBC

- Static call to class loader makes sure the drivers are loaded

- Static call to `DriverManager` goes across network and validates password

- Connection provides an environment to make a query

- Execute an SQL query, either select or update/insert/delete

- Return the ResultSet of the data that matches the query (select)

- In either a `Database` main, or JUnit harness, test the `Database` stand-alone:

```java
database db = new database("SuppDB", "4311", "4311"); // 5 lines of JDBC
resultSet rs = db.execute("select * from S");
try {
    while (rs.next())
        System.out.println(rs.getString("S_NO")); // 6th line of JDBC
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```
Main() should create n frames observing “s1”, likewise for “s2”
- java Obsmain 3 : means to create 3 frames observing s1, likewise for s2
- In one loop : create s1, create ObsFrame and give it s1; likewise for s2
- Supplier constructor stores number, creates db, calls loadSupplier() to load from DB
Supplier: Load Data

1: rs:=execute("select * from S where S.NO=" + number)
2: next()
3: name:=getString("SNAME")
4: status:=getString("STATUS")
5: city:=getString("CITY")

- Supplier calls loadSupplier() at construction to load from DB
- loadSupplier() executes a query to locate particular S.NO
- Note that query needs to have embedded quotes around the number
  "select * from S where S.NO='" + number + '""
- next() increments the cursor forward to the one selected row (6th line of JDBC)
- getString() loads the values of the fields (7th line of JDBC)
Supplier: Store Data

1: number:=aNumber
2: name:=aName
3: status:=aStatus
4: city:= aCity
5: setChanged()
6: notifyObservers()
7: storeSupplier()

- ObsFrame calls updateSupplier() when the button is pushed
- Supplier’s mutator method sets the fields
- Call Observable’s setChanged() to set the “dirty” bit
- Call Observable’s notifyObservers to iterate through Observers (only if bit is set)
- Call storeSupplier() to update fields in database

execute("update S set SNAME="+name+
" , STATUS="+status+
" , CITY="+city+
" where S_NO="+number

- Supplier calls storeSupplier() whenever the Supplier mutates
- Update the fields in the database
- name, city, number require quotes
- Use embedded single quotes
• ObsFrame calls displaySupplier() at construction and when notified (update) by Observable

• ObsFrame’s displaySupplier() accesses each Supplier field, and displays
• AWT calls actionPerformed() when the button (or menu) is selected

• ObsFrame “is a” ActionListener and must have actionPerformed()

• Convert the event into a String argument

• If ”Exit”, shutdown the System

• Sample each TextField and hand-over to the Supplier’s mutator

• Note that a,b,c,d are just placeholders and do NOT need to be used
• Register this ObsFrame with the Supplier (don’t forget to save the Supplier)

• Make a FlowLayout

• Add Labels

• Add TextFields

• Create a SUBMIT or UPDATE button

• ObsFrame listens for a push on a button

• Make a Menu

• Call displaySupplier() to load initial data
• Use Java’s serialization mechanism to perform Snapshot Pattern for persistence

• Alternative to object persistence using SQL

• loadSupplier(), storeSupplier() use serialization instead of SQL database

• Each Supplier record is store in it’s own file (e.g. “S1.dat”)

• Class must implement the Serializable interface, but requires no extra methods

• The class is “marked” as a candidate for serialization

• Supplier attributes moved to Supp class

• DataBase handle replaced by Supp handle

• Memory to disk: (new ObjectOutputStream(new FileOutputStream(filename))).writeObject(obj);

• Disk to memory: obj = (new ObjectInputStream(new FileInputStream(filename))).readObject();

• File Streams “know” how to read/write bytes of a file

• Object Streams “know” the attributes/methods of the class, and use the File Streams

• ObjectOutputStream completely serializes (flattens) the Supplier object

• Data file contains enough information to completely reconstruct the object in memory

• import java.io.*;
Snapshot Pattern: Serialization

```
<< interface >>
Serializable

Marker Pattern

Supplier     suppl
Supp
+number:String
+name:String
+status:String
+city:String
Supp(aNumber:String)

create(aNumber) :Supplier
1: supp:=create(aNumber)
2: loadSupplier()

loadSupplier() :Supplier
1: f:=create(supp.number+".dat")
2: in:=create(f)
3: supp:=readObject()
4: close()

storeSupplier() :Supplier
1: f:=create(supp.number+".dat")
2: out:=create(f)
3: writeObject(supp)
4: close()
```
Recall the Remote Proxy Pattern to achieve Distributed Objects on the network

Java’s Remote Method Invocation (RMI) implements this pattern

The Proxy is called a “Stub”

The RealSubject has a “Skeleton” that talks across the network to the Stub

Both are automatically produced by the rmic compiler

The RealSubject is named an “Impl”

Both the Stub and the Impl implement the same interface (e.g. Product)

This guarantees that they have the same methods, hence the same look-and-feel (e.g. get())

The Client has a variable of type Product interface, and makes ordinary calls

But the variable will actually be the instantiation of the Stub

Any call to the Stub will get sent (serialized) across the network to the Skeleton

Skeleton makes actual call on Impl, then returns any (serialized) results to the Stub

The Client thinks it is dealing directly with the Impl, as if it were on the same computer

The registry program is started in the background: rmiregistry &

The Server is started in the background: java ProductServer &

The Server news the Impl, and registers it

This allows many Client programs to interact with the distributed object: java ProductClient
• import java.rmi.* provides a hierarchy of classes

• Programmer only provides ProductImpl application and Product interface

• Both Stub and Impl implement the same Product interface

• Stub and Skeleton are automatically produced: rmic ProductImpl
• Next page provides an example Product application

• Product interface only has a getDescription() method, which throws an exception

• ProductImpl extends UnicastRemoteObject, implements Product

• Constructor throws an exception

• Stub and Skeleton are automatically produced: rmic ProductImpl

• ProductServer news two Impls and registers with Naming.rebind() using a primary key

• ProductServer runs in the background

• ProductClient locates the Skeleton with Naming.lookup() using URL and primary key

• The lookup instantiates the Stub, which the Client uses
// Product.java
import java.rmi.*;
interface Product extends Remote {
    public String getDescription() throws RemoteException;
}
// ProductImpl.java
// javac ProductImpl.java
// rmic ProductImpl (=> ProductImpl_Skel.class, ProductImpl_Stub.class)
import java.rmi.*;
import java.rmi.server.*;
public class ProductImpl extends UnicastRemoteObject implements Product {
    private String descr;
    public ProductImpl(String d) throws RemoteException { descr = d; }
    public String getDescription() {
        return "I am a " + descr + ". Buy me!";
    }
}

//ProductServer.java
//To run: rlogin snazzy          (for example)
//      rmiregistry &          (one time only)
//     java ProductServer &
import java.rmi.*;
import java.rmi.server.*;
public class ProductServer {
    public static void main(String srgs[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            ProductImpl p1 = new ProductImpl("Blackwell Toaster");
            ProductImpl p2 = new ProductImpl("ZapXress Microwave Oven");
            Naming.rebind("toaster", p1);
            Naming.rebind("microwave", p2);
        } catch (Exception e) { System.out.println("Error: "+e); }
    }
}

// ProductClient.java - run on local machine after starting server
import java.rmi.*;
import java.rmi.server.*;
public class ProductClient {
    public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        String url = "rmi://gold/";     // for example
        try {
            Product c1 = (Product)Naming.lookup(url+"toaster");
            Product c2 = (Product)Naming.lookup(url+"microwave");
            System.out.println(c1.getDescription());
            System.out.println(c2.getDescription());
        } catch (Exception e) { System.out.println("Error: "+e); }
    }
}
- Supplier application should be converted to RMI (see Exercise)
- Server X is started in the background
- Server creates an Impl, registers with Naming, and has a Skeleton
- Client runs, looks up the registered Impl using Naming, gets a Stub
- Client uses Stub to mutate, which goes across the network to the Skeleton
- Skeleton calls mutate on Impl, which then updates the database/file on disk
• Use your Supplier application, with either SQL or Serialize for persistence as usual

• Supplier method signatures go into a Supplier interface, which extends Remote

• Rename Supplier as SupplierImpl

• SupplierImpl extends UnicastRemoteObject (not Observable anymore), implements Supplier

• Put exceptions on interface methods and SupplierImpl constructor

• Modify ProductServer to be SupplierServer

• Modify ProductClient to be SupplierClient

• Test:

```
unix% rmiregistry &
unix% java SupplierServer &
unix% java SupplierClient

msdos% start rmiregistry
msdos% start java SupplierServer
msdos% java SupplierClient
```
Part V
Persistence Framework

- application: Point-Of-Sale Terminal (POST)
- keep data persistent in a database
- but use delayed instantiation
- only materialize from database when needed
- use virtual proxies for this delay
- proxy feels like the real subject, same interface
- but virtual will delay the instantiation
- materialization occurs when data/code really needed
- dematerialization (write to database) occurs for modified (dirty) data
- but is only performed with a commit from the client
- all modification effects are undone if client does a rollback
public class ProductSpecification {
    private String description;
    private double price;
    private int upc;
    public ProductSpecification(String description, double price, int upc) {
        this.upc = upc;
        this.price = price;
        this.description = description;
    }
    public String getDescription() { return description; }
    public double getPrice() { return price; }
    public int getUpc() { return upc; }
    public void setDescription(String description) { this.description = description; }
    public void setPrice(double price) { this.price = price; }
    public void setUpc(int upc) { this.upc = upc; }
}
import java.util.Vector;
public class Sale {
    private String saledate;
    private String saletime;
    private Vector lineItems = new Vector();
    public Sale(String saledate, String saletime) {
        this.saledate = saledate;
        this.saletime = saletime;
    }
    public void setDateTime(String saledate, String saletime) {
        this.saledate = saledate;
        this.saletime = saletime;
    }
    public void makeLineItem(SalesLineItem item) { lineItems.addElement(item); }
    public double total() {
        double sum = 0;
        for (int i=0; i<lineItems.size(); i++) {
            SalesLineItem sli = (SalesLineItem) lineItems.elementAt(i);
            sum += sli.subtotal();
        }
        return sum;
    }
}
public class SalesLineItem {
    private ProductSpecification ps;
    private int qty;
    public SalesLineItem(ProductSpecification ps, int qty) {
        this.ps = ps;
        this.qty = qty;
    }
    public void setQty(int qty) { this.qty = qty; }
    public double subtotal() { return qty * ps.getPrice(); }
}
TELNET TO UNIX.

source /usr/local/vcs/README
mysql -p -u 6310 -h gold postDB
Enter password: 6310
mysql> select * from Products;

Products:
+-----------------+-----------+-----+-------+
| OID  | description | price | upc   |
+-----------------+-----------+-----+-------+
| p1   | tofu       | 10.00| 111   |
| p2   | tempeh     | 2.25 | 222   |
| xyz1000 | apples   | 5.00 | 333   |
+-----------------+-----------+-----+-------+

Sales:
+-----------------+----------+----------+
| OID  | saledate | saletime |
+-----------------+----------+----------+
| xyz123 | 01/01/1997 | 10:00    |
| abc345 | 02/02/1997 | 14:00    |
+-----------------+----------+----------+

SalesItems:
+-----------------+----------+
| SALEDID | ITEMOID |
+-----------------+----------+
| xyz123  | sli1    |
| xyz123  | sli2    |
+-----------------+----------+

Items:
+-----------------+----------+----------+
| OID  | qty | PSOID |
+-----------------+----------+----------+
| sli1  | 1   | p1     |
| sli2  | 2   | p2     |
+-----------------+----------+----------+

- OID primary key uniquely identifies each object
- SalesItems is an intersection table between Sales and Items
- Sales has many Items
- Item belongs to only one Sale
- intersections are necessary when there is a many-to-many relationship
- but using SalesItems anyway, to make application more challenging
Virtual Proxy:
- oid: String
- realSubject: Object
- getRealSubject(): Object
- materializeSubject()

ProxyInterface:
- request()

Client:
    concreteVirtualProxy.request()

ConcreteVirtualProxy:
1: realSubject = getRealSubject()
2: realSubject.request()

VirtualProxy:
1.1: if (realSubject not materialized)
    materializeSubject()
    return realSubject

RealSubject:
2.1: perform actual request using real data

- Proxy Pattern: Concrete is a Proxy for Real
- Concrete and Real implement the exact same interface, same feel
- Virtual only knows about Objects in general
- Concrete knows about the Real specifically, designed as a match
- Virtual has an OID as primary key to Real
- Virtual has a handle to Real, but delays the instantiation
- Concrete can ask for Real handle, which then cause instantiation
Client:
    productSpecificationProxy.getPrice()

ProductSpecificationProxy:
    1: realSubject = getRealSubject()
    2: return realSubject.getPrice()

VirtualProxy:
    1.1: if (realSubject not materialized)
         materializeSubject()
         return realSubject

RealSubject:
    2.1: return price

ALTERNATIVELY:

Client:
    salesLineItem.subtotal()

SalesLineItem:
    1: return quantity * ips.getPrice()
- Items would also have a Broker
- Top Layer: object cache management
- Middle Layer: query management
- Relational used for databases
- File used for Unix flat file
- Can have many different mechanisms at middle layer
- Bottom Layer: application specific, one broker per class
- Each is implemented as a Singleton
Virtual Proxy HAS-A Broker

VirtualProxy

oid: String
realSubject: Object
broker: PFWBroker

getRealSubject: Object
materializeSubject():
createBroker(): PFWBroker
gerBroker(): PFWBroker

materializes-from

PFWBroker

objectWith(anOID): Object

File

PS Sale SLI

Client:
productSpecificationProxy.getPrice()

ProductSpecificationProxy:
1: realSubject = getRealSubject()
1.1.1.1: return ProductSpecificationRelationalBroker.getInstance() // SINGLETON
2: return realSubject.getPrice()

VirtualProxy:
1.1: if (realSubject not materialized)
    materializeSubject()
    return realSubject
1.1.1: broker = getBroker()
1.1.1.1: if (broker not created)
    broker = createBroker()
    return broker
1.1.2: realSubject = broker.objectWith(oid)

ProductSpecification:
2.1: return price

• abstract getBroker is Template Method Pattern
• but since it is used for instantiation, it is Factory Method Pattern
objectWith:
  obj = inCache(anOID)
  if (obj not NIL)  // is in cache
    return obj
  else
    return materializeWith(anOID)

- cache stores objects based on a key
- PS Broker is a Singleton, therefore, one Cache of all Product Specifications
Broker Hierarchy

**PFWBroker**
- cache: ObjectCache
- objectWith(anOID): Object
- inCache(anOID): Object
- addOldClean(anOID, obj)
- materializeWith(anOID): Object

**RelationalPFWBroker**
- currentRecordAsObject(rs): Object
- materializeWith(anOID): Object
- selectFirst(anOID): ResultSet

**ProductSpecificationRelationalBroker**
- currentRecordAsObject(rs): ProductSpecification

**PFWBroker:**
materializeWith(anOID)

**RelationalPFWBroker:**
1: rs = selectFirst(anOID)
2: obj = currentRecordAsObject(rs)
3: addOldClean(anOID, obj)
4: return obj

1.1: return db.execute("select * from tableName where OID = anOID")

**ProductSpecificationRelationalBroker:**
2.1: return new ProductSpecification(
    rs.getString("description"),
    rs.getDouble("price"),
    rs.getInteger("upc")
)

- how to access the database?
- what tableName? who knows? how to access?
- abstract methods force specialization
- abstract is the Template Method Pattern
- used for instantiation means it is specifically Factory Method
- middle layer uses code above, within, and below
- good example of generalization vs. specialization
Client Program

```java
ps = new ProductSpecificationProxy("p1") // only makes a proxy
println ps.getDescription() // triggers database access (materialization)
println ps.getDescription() // proxy has handle to realSubject
ps1 = new ProductSpecificationProxy("p1") // another proxy
println p1.getDescription() // can find object in cache
```
- Broker Server is a Singleton
- Brokers register themselves with the Broker Server at construction time
- Mutator methods cause dirty on the Broker, hence dirty on the Cache
- Clients commit/rollback on the Broker Server
- Broker Server commits on every registered Broker
- Each Broker commits on its own Cache
Marked Brokers

BrokerServer:

- Singleton Pattern with a commit.

Brokers:

- All register themselves with the BrokerServer.
- All have a commit.

Marked Brokers:

- Three classes of Brokers (X, Y, Z) are specialized with their own versions of commit.
- In this case, commit just does a println to identify the class.
- The specialized Brokers fall into two "marked" categories: Red and Blue.
- BrokerX and Y are Red. BrokerZ is Blue.

Client:

- Instantiates one instance of BrokerX, Y, Z.
- Asks the BrokerServer to do a commit, but the server only notifies the Red category.

Exercise:

- Draw UML class diagrams.
- Draw UML collaboration diagrams.
- Code in Java.
Commit/Rollback

<table>
<thead>
<tr>
<th>State</th>
<th>Commit:</th>
<th>Rollback:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cache</td>
<td>DB</td>
</tr>
<tr>
<td>NewClean</td>
<td>OldClean</td>
<td>insert</td>
</tr>
<tr>
<td>OldClean</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>NewDirty</td>
<td>OldClean</td>
<td>insert</td>
</tr>
<tr>
<td>OldDirty</td>
<td>OldClean</td>
<td>update</td>
</tr>
<tr>
<td>NewDeleted</td>
<td>remove</td>
<td>*</td>
</tr>
<tr>
<td>OldDeleted</td>
<td>remove</td>
<td>delete</td>
</tr>
</tbody>
</table>

Materialize:
- rs = SQL execute(query)
- currentRecordAsObject(rs)
- new ProductSpecification(rs.getString('description'))

Dematerialize:
- if (deleteFlag) SQL execute('delete from')
- if (insertFlag) SQL execute('insert into')
- insert is FALSE, TRUE
- delete is TRUE, FALSE
- update is TRUE, TRUE
- objectAsRecord returns the SQL insert string

Test Program:
- new 'team3'
- get tofu
- new 'team3'
- get tofu
- set XXX
- rollback
- get tofu (But here is a problem. See your checkoff sheet)
- set TOFU
- commit
- verify by running again or using MySQL to examine DB
Part VI
Tcl

29 Tcl Notes

Tcl/Tk

- created by John Ousterhout, Berkeley, now Sun
- Tcl: Tool command language, a script interpreter
- Tk: extension Toolkit to interface to X (Motif)
- can be used to make entire applications
- can be used just for GUI
- easy to make prototypes
- many types of widgets
- canvas, button, frame, scroll, text, entries
- event-driven
- commercial products and 10,000+ users
- excellent news group
- Unix, Windows, Macintosh, VMS, Next, OSF/1

Tcl: A script language

% wish

   interactive session with shell and widgets.

   % source myfile

% tclsh

   interactive session with shell - no widgets.

% ./myfile

   #!/usr/local/bin/wish -f
   # put your Tcl commands in your executable file

Special Characters:

$  variable name substitution   $a,$array(one),$array($a)
Space separates list elements   one
two
tree
" " quote string: $ substitution puts "results = $res"
{} NO $ substitution           puts {use $res for value}
;   terminates command          set a 1; set b 2
\   single character quoting   puts this\ is\ one\ string
#   treat line as a comment     set a 1; # initialize a
Tcl: variables and expressions

% tclsh
% set a 24
24
% set b $a
24
% set b [expr $a/2]
12
% set c hello
hello
% set c "hello"
hello
% set c "a string"
a string
% set c "a string with $a"
a string with 24
% set c "a string with \$a"
a string with $a
% set c {a string with $a}
a string with $a
% set c {set a $b}
set a $b
% eval $c
12

Tcl: Associative Arrays

% tclsh
% set x(1) 5
5
% set x(one) 3
3
% set x(3) one
one
% set x(1,3) blue
blue
% set x(blue) 1,3
1,3
% array size x
5
% array names x
blue 1,3 1 one 3
% set y $x(3)
one
%set i 3
3
% set y $x($i)
one
%tclsh
% set L {red blue green}
red blue green
% lindex $L 2
green
% set L [linsert $L 2 yellow]
red blue yellow green
% llength $L
4
% lrange $L 1 3
blue yellow green
% lrange $L 1 end
blue yellow green
% set L [lrange $L 1 end]
blue yellow green
% lsearch -exact $L yellow
1
% lsearch -glob $L yel*
1
% lsearch $L black
-1
% set L [concat $L {purple pink}]
blue yellow green purple pink

Tcl: Strings

% set s "here is a string"
here is a string
% string compare $s "here is another string"
-1
% string first "is" $s
5
% string index $s 5
i
% string length $s
16
% string range $s 0 3
here
% string range $s 10 end
string
% llength $s
4
% lindex $s 0
here
% lindex $s 3
string
Tcl: Loops (FOR, FOREACH, WHILE)

for {set i 1} {$i<=10} {incr i} {
    puts $i
}

foreach i {1 2 3 4 5 6 7 8 9 10} {
    puts $i
}

set i 1
while {$i <= 10} {
    puts $i
    incr $i
}

All three produce the same output (1, 2, ..., 10).

Tcl: Scripts as Lists and Strings

% set s [list for {set i 1} {$i<=5} {incr i} {puts $i}]
for {set i 1} {$i<=5} {incr i} {puts $i}
% llength $s
5
% eval $s
# all FOR loops have 5 list elements
1
2
3
4
5
% foreach j $s {puts $j}
for
set i 1
$i<=5
incr i
puts $i
% set k [lindex $s 3]
incr i
% lindex $k 0
incr
% string index $k 3
# k can be treated as a string
Tcl: Branching (IF, SWITCH)

% set i 1
1
% if {$i > 0} {
  puts "positive $i"
} else {
  puts "nonpositive $i"
%
} positive 1
% switch $i {
  0 {puts zero}
  1 {puts one}
  blue {puts blue}
  green {puts $i}
%
} one

IF statements have 3 or 5 list elements.
SWITCH statements have 3 list elements.
FOR loops have 5 list elements
FOREACH has 4 list elements
WHILE has 3 list elements

Tcl: Procedures

PROCEDURES have 4 list elements:

% set x 10
10
% proc max {a b} {
  global x
  puts "x = $x"
  if {$a > $b} {
    puts "$a is greater than $b"
  } else {
    puts "$a is not greater than $b"
  }
%
} max 3 5
x = 10
3 is not greater than 5

Variables (parameters, globals, locals) are “typeless” (strings).
Tcl: Files
Read each line in a file as a string:

```tcl
set f [open "myfile" r]
while {[gets $f line] >= 0} {
    puts $line
}
close $f
```

Read entire file as one string (or list or script):

```tcl
set f [open $name r]
set s [read $f]
close $f
```

Tcl as Structured Pseudocode

```tcl
proc example {some arguments} {
    initialize and open file
    while {there is data left in the file} {
        get next record
        if {value is out of range} {
            raise exception
        } else {
            process record
            display results
        }
    }
    terminate and close file
}
```

The list structure of the pseudocode makes for easy parsing by Tcl.
Example: element #4 of the IF statement is the body of the ELSE
Tcl String Commands

#!/opt/local/bin/wish -f
wm withdraw . ;# OUTPUT

set a "abcdef"

puts $a ;# abcdef

puts [string length $a] ;# 6

puts [string compare $a "boy"] ;# -1

puts [string first "cde" $a] ;# 2

puts [string index $a 2] ;# c

puts [string match abcdef $a] ;# 1

puts [string match cde $a] ;# 0

puts [string match *cde* $a] ;# 1

puts [string range $a 2 4] ;# cde

puts [string toupper $a] ;# ABCDEF

puts [length $a] ;# 1

exit
#!/usr/local/bin/wish -f
wm withdraw . ;# OUTPUT
set a [list a b c]
set b [list d e f]
set c [concat $a $b]
puts $c ;# a b c d e f
set a "$a $b"
puts $a ;# a b c d e f
set b [join $a []] puts $b ;# a|b|c|d|e|f
set b [lappend a h i j] puts $b ;# a b c d e f h i j
puts $a ;# a b c d e f h i j
puts [lindex $a 2] ;# c
set a [linsert $a 3 x y z]
puts $a ;# a b c x y z d e f h i j
puts [llength $a] ;# 12
puts [lrange $a 3 5] ;# x y z
set a [lreplace $a 3 5 X Y Z]
puts $a ;# a b c X Y Z d e f h i j
puts [lsearch -exact $a X] ;# 3
set b [a b {puts hello} c]
puts $b ;# a b {puts hello} c
set b [list a b {puts hello} c]
puts $b ;# a b {puts hello} c
puts [llength $b] ;# 4
set c [lrange $b 2 2]
puts $c ;# {puts hello}
set d [lindex $b 2]
puts $d ;# puts hello
eval $d ;# hello
exit
#!/usr/local/bin/tclsh
wm withdraw .

set a puts
set b HELLO
set c GOODBYE
eval \{a $b; $a $c\}

set xarray(1) \{HELLO $yarray(z)\}
set yarray(z) GOODBYE

proc valid1 {} {
    global xarray yarray
    set out1 [lindex xarray(1) 0]
    set out2 [lindex xarray(1) 1]
    puts $out1 ;# HELLO
    puts $out2 ;# $yarray(z)
}
valid1

proc valid2 {} {
    global xarray yarray
    eval \{set out1 [lindex xarray(1) 0]\}
    eval \{set out2 [lindex xarray(1) 1]\}
    puts $out1 ;# HELLO
    puts $out2 ;# $yarray(z)
}
valid2

proc valid3 {} {
    global xarray yarray
    eval eval \{set out1 [lindex xarray(1) 0]\}
    eval eval \{set out2 [lindex xarray(1) 1]\}
    puts $out1 ;# HELLO
    puts $out2 ;# GOODBYE
}
valid3

exit
# execute a C program from a Tcl script
exec myCprogram

# and capture its output directly
set results [exec myCprogram]

# or redirect its input/output
exec myCprogram <myinput.txt >myoutput.txt

# or run the Java interpreter
set results [exec java myJavaprogram]

# execute any OS command: ls, rm, cat, grep, wc, etc
# grep looks for occurences of the exec command in glob.tcl
puts [exec grep 'exec' glob.tcl]

#Tcl glob command lists names of files/directories using wildcards
set files [glob *]
puts $files

# look at each file name on the directory
foreach f $files {
    # Tcl file command has many options: type, isDirectory, readable, owned
    puts [file type $f]
    # examine the word count on each file
    puts [exec wc $f]
    # print each file but directories cause an exception - so catch it
    catch {set src [exec cat $f]; puts $src}
}
# Tcl upvar command allows Call-By-Reference parameters.
# arrays don’t have values as a whole: $myarray is not a legal value.
# usually use upvar to access individual elements of an array.

proc array name {

    # assign (alias) the array to a local variable 'a'
    upvar $name a

    # get the names (indices) of the array and sort them
    foreach el [lsort [array names a]] {
        # print out the values of each element
        puts "$el = $a($el)"
    }
}

set info(age) 37
set info(position) "Vice President"
parray info

My Own Switch Method

#!/usr/local/bin/wish -f
wm withdraw .

proc myswitch {case body} {
    for {set i 0} {$i < [llength $body]} {incr i; incr i} {
        if {[string compare $case [lindex $body $i]] == 0} {
            eval [lindex $body [expr $i + 1]]
            break
        }
    }

    set i 0

    myswitch $i {
        0 {puts zero}
        1 {puts one}
        blue {puts blue}
        green {puts $i}
    }
    exit
GOAL:

Design and code a function-oriented Tcl script to perform reverse engineering on the object-oriented input. That is, extract the design of the classes from the Java source code. Output in textual or graphical format. Key issues are parsing, lists, and trees.

INPUT: Java Source Code (syntax might be subset of full language)

OUTPUT: Object-Oriented Design

- Classes
- Attributes (which may be other objects, i.e. hierarchy of assembly)
- Methods
- Inheritance

NOTES:

- Write a suitable Java test program as you go along.
- Read in entire source code at once (all classes in the same file).
- Call parse functions based on known structure of Java.
- Actual method code is irrelevant and can be easily discarded in Tcl.
- Store accumulated information in Tcl data structures.
- Print out summary information in an HTML file similar to Javadoc.
public class Test extends Thread {

    public Test () {
    }

    public int op2 () {
        return 0;
    }
}

// javadoc Test.java
// ABBREVIATED HTML FILE:

<html>
<head>
<title>
    Class Test
</title>
</head>
<body>
<h1>
    Class Test
</h1>
<pre>
java.lang.Object
 |
    +----java.lang.Thread
      |
        +----Test
</pre>
<hr>
<dl>
    <dt> public class <b>Test</b> 
    <dt> extends Thread
</dl>
<b>Test</b>
<pre>
    public Test()
</pre>
<pre>
    public int op2()
</pre>
</body>
</html>
Introduction

What is Tcl? A scripting language that may be invoked the same way any Unix shell scripting language
with a #! in the first line and the path to tclsh. What is Tk? An extension to tclsh (invoked as wish)
that is a toolkit of Motif-like widgets for use in creating GUIs.
The Tcl/Tk language differs from other scripting languages in that it is event driven (i.e., push a
button, something happens), asynchronous, and has procedural constructs so that structured code can
be created.
Tcl/Tk applications can be written entirely in Tcl/Tk (this is particularly useful as compilation is not
necessary). Such applications can be made extremely powerful and can still make use of the speed of
compiled code (such as C) when Tcl executes programs or communicates via a pipe with a program.
On the other hand, a C program can be written that starts a Tcl interpreter, where Tcl/Tk library
calls can be made.
Some simple ideas about the Tcl language:
- There are no types in Tcl: everything is considered to be a string. This makes for speedy and simple
coding, however this can contribute to confusion when dealing with math (even though Tcl has a
collection of built-in math functions).
- Everything between {} is considered to be an individual Tcl script. This idea is particularly useful
when parts of code are desired to be parsed and executed at run time and not when the Tcl code is
first loaded (or sourced). Code executed between {} produces a result of 1 or 0 (true or false).
- Code executed between || returns a value, such as the return value of a Tcl or user function.
- Double quotes, " " or " ", enclose strings separated by white space that are to be viewed by Tcl as
one string. Variable substitution occurs immediately for variables referenced inside double quotes.
- $ causes variable substitution.
Most of the examples in these notes can be run inside the shell that is given to you when you invoke
"wish". (If an example does not work, exit wish and restart it.)

An Overview of the Tcl Language

To create a variable use the set command...

```
set a hello
>hello

... Tcl responds with "hello". To check the value in the variable again, preface it with $, and send it
to standard output with the puts command...
```

```
puts $a
>hello
```

Variables can be local or global depending on where the first set statement occurs. In the following
examples, proc is the keyword for a procedure.

```
set a hello
proc b {} {
  set a goodbye
  puts $a
}
```
To invoke the procedure, type b, and you will get the answer on standard out:
>goodbye
The following ....

set a hello
proc b {} {
global a
    puts $a
}
b

... produces the output on standard out: >hello
This one will produce an error...

set a hello
proc b {} {
    puts $a
}
b
>can't read "a": no such variable

Tcl errors are of a stack trace variety. Most often, Tcl will produce several lines of errors. The first line is usually all one needs to examine in order to understand the error. The remaining lines are a trace backwards to help the programmer locate the source line that is in error.
Math in Tcl is accomplished via the expr keyword and a variety of math functions (sqrt, cos, round, log, etc.). So...

set a 5
set b [expr $a + 5]
puts $b
>10

...produces 10 on standard out. Note the use of [] brackets to return the value of the expr function.
Aside from the proc procedure construct, Tcl provides for other flow constructs as well.
Looping can be accomplished by these examples. First a for loop...

for {set i 1} {$i <= 3} {incr i 1} {
    puts $i
}
>1
>2
>3

A foreach loop executes for each member of a list...

foreach color {red green blue} {
    puts $color
}
>red
>green
>blue
A condition loop can be created using while...

```
set keepGoing 10
while {$keepGoing} {
    puts "All work and no play makes Jack a dull boy."
    incr keepGoing -1
}
```

... note the use of incr (increment) to decrement the keepGoing variable.
In event programming a common use of the while loop is to create an infinite loop...

```
while {1} {
    # When an event has occurred, such as the GUI user has pushed a
    # button, then break from this loop:
    if {$buttonPushed == 1} {
        puts "User has pushed a button"
        break
    } else {
        puts "Loop one more time."
    }
}
```

The above type of code segment (do not type it in) is often used to keep some action in the GUI going until the user commands otherwise. Note the use of an if-else statement, along with the fact that Tcl statements are separated by carriage return and line feed.
Along with the for, foreach, while, break and if-else constructs shown above, switch (case statement) and continue are also supported.
Data can be stored not only in single string variables, but in arrays and most handily in lists.
Arrays can be indexed by anything. So...

```
set anArray(1) first
set anArray(first) 1
```

... has created two elements of a single array and assigned values "first" and "1" to them.
Lists can be created by putting a set of elements between {} brackets...

```
set aList {red blue green}
puts $aList
>red blue green
```

Any variable, no matter how created, can utilize list processing utilities such as search, index, match, append, etc. This is also true of the string utilites (format, scan, compare, match, trim, etc.)
Tcl also provides for file operations in Tcl which are straight forward...

```
set fileId [open "~/.cshrc" r]
while {[gets $fileId line] > -1} {
    puts $line
}
close $fileId
```
... will open your .cshrc file in read mode and display its contents to standard output. The open command returns the file descriptor that Tcl uses to keep track of this open file. The gets command also returns a value indicating the length of each record it reads (-1 indicates EOF). Tcl File operations also include pipe I/O, character reading, writing and seek.
While Tcl statements can be typed interactively inside the interpreter (wish), most development is done using Tcl scripts that are created with a text editor. The scripts can then be executed by sourcing the script with wish. Many applications invoke wish with a command line argument that will load and execute the script named in the argument:  
\texttt{wish -f /home/student/jdoe/start.tcl}

Once a script has been loaded and a main procedure invoked from within the script, it is easy enough to source other scripts from inside the original starting script:  
\texttt{source /home/student/jdoe/otherstuff.tcl}

Another way to source in required Tcl procedures is to make use of a built in feature of Tcl called "auto loading". Whenever Tcl comes across a call to a procedure it has no knowledge of, it will search the pathnames listed in a Tcl system variable called auto_path. This path is used in a similar way to the PATH variable in Unix where the user can add to it at any time. The file that is being looked for is tclIndex, which contains the names of the user application’s procedures and the pathname to the Tcl script file that contains each procedure. Once the unknown procedure name is found in tclIndex, the associated script is automatically sourced. The command, auto_mkindex, will create the tclIndex file.