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

CS 4311 SOFTWARE ENGINEERING II

Introduction to the Unified Modeling Language and Design Patterns

Copyright 2006
Edward A. Billard
# Contents

## I Introduction to UML and Java

1 UML Notation .................................................. 4

2 UML Examples ................................................. 12

3 Exceptions .................................................. 19

4 Java Database Connectivity (JDBC) .............................. 22

## II Data Structures in UML and Java

5 Stack (LIFO) .................................................. 26

6 Queue (FIFO) .................................................. 28

7 Breadth-First Search (BFS) ................................... 33

8 Btree .................................................................. 35

## III Patterns

9 Singleton Pattern: One Global Object ............................ 52

10 Dynamic Starter Pattern: Runtime Decision ................. 54

11 Abstract Factory Pattern: Platform Independent GUI ....... 56

12 Object Pool Pattern: Reusable JDBC Connections .................. 58

13 Observer Pattern: Update Notification .......................... 60

14 Snapshot Pattern: Persistent Objects ........................... 62

15 Remote Proxy Pattern: Distributed Objects ...................... 63

16 Composite Pattern: JUnit Testing ............................... 64

## IV Case Study

17 Observable/Persistent Objects .................................... 74

18 Remote(RMI)/Persistent Objects ............................... 88
• 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
<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>-private Attribute</td>
</tr>
<tr>
<td>#protected Attribute</td>
</tr>
<tr>
<td>+public Attribute</td>
</tr>
<tr>
<td>class Attribute</td>
</tr>
<tr>
<td>method1()</td>
</tr>
<tr>
<td>method2(): type</td>
</tr>
<tr>
<td>method3(parameter: type)</td>
</tr>
<tr>
<td>-private Method()</td>
</tr>
<tr>
<td>#protected Method()</td>
</tr>
<tr>
<td>+public Method()</td>
</tr>
<tr>
<td>classMethod()</td>
</tr>
</tbody>
</table>

class ClassName {
    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:

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

Generalization/Specialization ("is a"):

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

Realization ("realizes"):

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

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

class Server {
    void serverOperation() {}
}
```

Composition (should be dark diamond):

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

- Composition creates/destroys sub-parts
- Part can only belong to one parent (Whole)
- Example: Car (Whole) to Carburetor (Part)
Aggregation (is a clear diamond):

```
class Container {
    java.util.Vector v;

    void addChild(Object o) { v.addElement(o); }
}
```

- Aggregation receives Objects from the outside
- Object can belong to more than one parent, and live longer than parent
- Use Vector (or Hash, etc.) to store collection of many Objects
- Example: Pond (Container) to Duck (Object)
- Composition is a stronger form of Aggregation

Multiplicity:

```java
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 {}
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());
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.

```
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() {}
}
Static Structure Diagram:

Collaboration Diagram:

- What is the Java code?
class A {
    private B b = new B();
    private C c = new C();
    public x() {
        b.y(c); // 1: is applied to b, and passes along c
        b.r();  // 2:
        c.s();  // 3:
        t();    // 4: is applied to self
    }
    private t() {
    }
}

class B {
    public y(C c) {
        c.z(this); // 1.1:
    }
    public q() {
    }
    public r() {
    }
}

class C {
    public z(B b) {
        b.q(); // 1.1.1:
    }
}

What is the Collaboration Diagram?
UML Example: Geometric Objects

Static Structure Diagram (Inheritance):

- 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++; // count number of objects
        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();
// 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());
        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 8 key lines of JDBC:

```java
import java.sql.*;
public class SuppDBquery {
    public static void main(String args[]) {
        try {
            // 1: load specific code to use JDBC
            Class.forName("org.sqlite.JDBC");

            // 2: establish connection to file (DB) using protocol, file location
            // This connection is for java files on ex1, ex2, etc.
            Connection con = DriverManager.getConnection("jdbc:sqlite:../sqlite/SuppDB");
            // This connection is for SuppDBquery located on the sqlite dir itself
            // Connection con = DriverManager.getConnection("jdbc:sqlite:SuppDB");

            // 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.getResultSet();

            // 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_NO") + " " + rs.getInt("STATUS"));
            }

            // 8: closes READ action, allows READ or WRITE action later
            rs.close();
        } catch (Exception e) {System.out.println(e.getMessage());}
    }
}
```
Java Database Connectivity (JDBC)

Uniform Resource Locator (URL):

SYNTAX: protocol, file location of DB

    String url = "jdbc:sqlite:../sqlite/SuppDB"

Note: DB is stored as ordinary file on sqlite directory, located ABOVE exercise directories.

SYNTAX: Class.forName(driverName)

    Class.forName("org.sqlite.JDBC");

Note: Drivers stored in jar file on sqlite directory.

Make a CONNECTION:

SYNTAX: DriverManager.getConnection(url)

    Connection con = DriverManager.getConnection(url);

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.getResultSet();

Get the METADATA (or DATA DICTIONARY):

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

    while(rs.next())

Get the COLUMN COUNT

    int n = md.getColumnCount();

Get the COLUMN NAME or INDEX:

    String name = md.getColumnName(i);
    int i = rs.findColumn("QTY");

Get the COLUMN TYPE or WIDTH:

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

Get the DATA by INDEX or NAME:

    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:

    rs.close();
    stmt.close();
    con.close();
Data Structures provide an opportunity to explore some initial software problems

Stacks and Queues can be designed in UML, implemented in Java

Breadth-First Search (BFS) may be performed on a tree implemented as a database

Btree provides an opportunity to re-engineering C code into Java code
5 Stack (LIFO)

Static Structure Diagram:

```plaintext
Stack
-STACK_SIZE:int=100
-EMPTY_TOS:int=-1
-top:int=EMPTY_TOS
-stack_array:int[STACK_SIZE]

+main(args: String[])
+Stack()
+isEmpty(): boolean
+isFull(): boolean
+push(i:int)
+pop(): int
```

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--]);
    }
}
Queue (FIFO)

Static Structure 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)

Collaboration Diagram:

- 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

Object

Object()
toString():String

Queue2

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

• 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
```

Collaboration Diagram:

```
main()

Queue2

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

<< new >>

Queue2

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

<< new >>

Integer

2: i:=create(5)

• 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;
        }
    }
}
```sql
sqlite> .quit
```

This table corresponds to this tree:

```
A
+------+
|      |
+------+
 |     |
|      |
B      C    D
+------+
 |     |
|      |
E      F    G    H
+------+
 |     |
|      |
I      J    K
```

and we would like a BFS or layered scan to produce this output:

```
A
B C D
E F G H
I J K
L
```
• Problem: Draw a layered tree starting from a root, say ”A” (parameter to program).

• How can you reuse your DataBase class from the Supplier case study?

• Alternatively, how can you use the code given in the JDBC tutorial?

• What SQL do you need?

• What does your data structures book say about BFS?

• You don’t need to actually build the tree itself in Java.

• This is not a recursion problem.

• How can you know the depth of a node?

• How do you know that the tree is on a new level?

• What should you store in a node?

• Can use Java’s Vector or one of the Queue classes.

• Another data file for SQL: Offspring (the Kings/Queens of England)

BFS Pseudo-code:

    enqueue the root node

    while the queue is not empty do {
        dequeue a node
        display the node
        for all children of the node do {
            enqueue the child node
        }
    }

• Using the tree on the previous page and the code above, what does the Queue look like over time?
- Binary Search Trees allow for fast searching: if key is smaller, go left. If key is larger, go right.

- Btrees use the same principle but are not Binary, they are Balanced

- Each node (page) can split into many directions, say up to 5

- The keys provide a “roadmap” to locate a given key
**Order**: 5 (max 5 children/node; max 4 keys/node; min 2 keys/node)

**Find**: h, x

```
ab   de   ghi   klmn   stx
```

**Insert**: u (OK)

```
ab   de   ghi   klmn   stux
```

**Insert**: p (Overflow)

```
ab   de   ghi   klmnp   stux
```

**Split; Promote m; Overflow**

```
ab   de   ghi   kl   np   stux
```

**Split; Promote j; New Root**

```
cf   j    mr
ab   de   ghi   kl   np   stux
```
Delete: h (OK)

Delete: r (Not on leaf; Swap with successor on leaf s; Delete OK)

Delete: p (Underflow)

Borrow from neighbor; Moveleft s t
Delete: d (Underflow; Cannot borrow)

Combine a b c e; Remove old page; Underflow

Combine f j m t; Remove old root
Btree: Re-engineer C to Java

- The following is a correct C code implementation of a Btree
- Compile in C: gcc -o btree btree.c
- Test C: btree btree.dat
- Compile in Java: javac Btree.java
- Test Java: java Btree btree.dat
- Btree.java is partially converted from C
- Btree.java is mostly commented-out and does not actually do anything
- Translate all methods into Java

Hints:
1. start with method ListNode(), labeled “Convert #1”
2. move GIANT COMMENT before ListNode() to after ListNode()
3. when ListNode() compiles successfully, repeat for “Convert #2”, ...
4. each method should be static
5. malloc: new
6. printf: io.print, io.println
6a. printf("%c",ch): io.print(ch+""), printf("%c %c," ,ch1,ch2): io.print(ch1+" +ch2+" ,")
7. *char: String
8. p->key: p.key
9. parameter/variable Node_type *p is just an object reference: Node_type p
10. parameter int *k is call-by-reference: how can Java do this in a good OO way? (also char *c)
11. parameter Node_type **p is call-by-reference, and can be solved according to 10
12. free should not be used, let Java do garbage collection
/* Btree.java

source : Btree.java
compile: javac Btree.java
run : java Btree <btree.dat
NOTE : needs IO.java and IO.class from same directory

compare results with C version:

source : btree.c
run : btree <btree.dat

Assignment:

Convert the commented-out C code here into Java.
Uncomment as you go along.
Search on Convert # for recommended order of conversion
*/

/* C version
typedef struct node_tag{
    int count;
    Key_type key[MAX+1];
    struct node_tag *branch[MAX+1];
}Node_type;
*/

// Converted
class Node_type {
    public int count;
    public char key[] = new char[Btree.MAX+1];
    public Node_type branch[] = new Node_type[Btree.MAX+1];
};

class Btree {

    // Converted
    public static void main (String args[]) {

        if (args.length == 0)
            io = new IO(System.in,System.out);
       else
            io = new IO(args[0],System.out);
        interact();

    }

    static final int MAX = 4;
    static final int MIN = 2;
    static final int MAXD = 10;
    static final int MAXWD = 50;
// typedef char Key_type; just use char in Java

/* C version
int step=0;
int cnt[MAXD];
int levels;
int loc[MAXD][MAXWD];
Node_type *ptrs[MAXD][MAXWD];
Node_type *root;
int silent = 0;
FILE *fp;
*/

// Converted
static int step=0;
static int cnt[] = new int[MAXD];
static int levels;
static int loc[][] = new int[MAXD][MAXWD];
static Node_type [][][] = new Node_type[MAXD][MAXWD];
static Node_type root;
static IO io;

/* C version
void Error(char *msg)
{
    printf("%s
",msg);
}
*/

// Converted
static void Error(String msg)
{
    io.println(msg); // note that there is an io.print(String) also
}

/* C version
void DFS(Node_type *p, int depth)
{
    int i;

    if (p != NULL) {
        if (depth>levels) levels=depth;
        ptrs[depth][cnt[depth]] = p;
        cnt[depth]++;

        for (i=0; i<=p->count; i++)
            DFS(p->branch[i],depth+1);
    }
}
*/
// Converted
static void DFS(Node_type p, int depth)
{
    int i;
    if (p != null) {
        if (depth>levels) levels=depth;
        ptrs[depth][cnt[depth]] = p;
        cnt[depth]++;
        for (i=0; i<=p.count; i++)
            DFS(p.branch[i],depth+1);
    }
}

// Converted except to uncomment Delete and PrintAllNodes
static void interact()
{
    char key;
    root = null;

    for (; ; ) {
        io.print("> ");
        String s = io.getLine();
        if(s.length() > 0) {
            char cmd = s.charAt(0);
            switch (cmd) {
            case 'q' : return;
            case 'h' : io.println("h(elp; qUIT; iNSERT x; d(ELETE x; pRINT
                break;
                case 'i' : key = s.charAt(2);
                io.println("INSERT: "+ key);
                root=Insert(key,root);
                break;
                case 'd' : key = s.charAt(2);
                io.println("DELETE: "+key);
                // uncomment when Delete is converted
                // root=Delete(key,root);
                break;
                case 'p' : io.println("B-TREE:");
                // uncomment when PrinAllNodes is converted
                // PrintAllNodes(root);
                break;
            } // end switch
        } // end if
    } // end for
}
/*
BEGIN: ONE GIANT COMMENT

// Convert #1
void ListNode(int depth,int cnt,Node_type *p)
{
    int i;

    printf("%d %d ",depth,cnt);
    for (i=1; i<=p->count; i++)
        printf("%c ",p->key[i]);
    printf("\n");
}

// Convert #2
void SetLoc()
{
    int i,depth,pos,base,first,last,dist,x;

    depth = levels;
    pos = 0;
    for (i=0; i<cnt[depth]; i++)
    {
        loc[depth][i] = pos;
        pos += (ptrs[depth][i]->count)*2+2;
    }
    for (depth=levels-1; depth>=0; depth--)
    {
        base = 0;
        for (i=0; i<cnt[depth]; i++)
        {
            first = base;
            last = first+(ptrs[depth][i]->count);
            dist = (loc[depth+1][last] - loc[depth+1][first] + 2*ptrs[depth+1][last]->count)/2;
            loc[depth][i] = loc[depth+1][first]+dist-ptrs[depth][i]->count;
            base += ptrs[depth][i]->count+1;
        }
    }
}

// Convert #3
void MoveRight(Node_type *p,int k)
{
    int c;
    Node_type *t;

    t = p->branch[k];
    for (c=t->count; c>0; c--){
        t->key[c+1]=t->key[c];
        t->branch[c+1]=t->branch[c];
    }
    t->branch[1] = t->branch[0];
    t->count++;
    t->key[1]=p->key[k];
t = p->branch[k-1];
p->key[k] = t->key[t->count];
p->branch[k] = q->branch[0];
t->count--;
}

// Convert #4
void MoveLeft(Node_type *p, int k)
{
    int c;
    Node_type *t;

    t = p->branch[k-1];
t->count++;
t->key[t->count] = p->key[k];
t->branch[t->count] = p->branch[k]->branch[0];
t = p->branch[k];
p->key[k] = t->key[1];
t->branch[0] = t->branch[1];
t->count--;
    for (c=1; c<=t->count; c++){
        t->key[c] = t->key[c+1];
t->branch[c] = t->branch[c+1];
    }
}

// Convert #5
void Combine(Node_type *p, int k)
{
    int c;
    Node_type *q;
    Node_type *l;

    q = p->branch[k];
l = p->branch[k-1];
l->count++;
l->key[l->count] = p->key[k];
l->branch[l->count] = q->branch[0];
    for (c=1; c<=q->count; c++){
        l->count++;
l->key[l->count] = q->key[c];
l->branch[l->count] = q->branch[c];
    }
    for (c=k; c<p->count; c++){
        p->key[c] = p->key[c+1];
p->branch[c] = p->branch[c+1];
    }
p->count--;
    free(q);
}
```c
void Restore(Node_type *p, int k)
{
    if (k==0)
        if (p->branch[1]->count > MIN)
            MoveLeft(p,1);
        else
            Combine(p,1);
    else
        if (k==p->count)
            if (p->branch[k-1]->count > MIN)
                MoveRight(p,k);
            else
                Combine(p,k);
        else
            if (p->branch[k-1]->count > MIN)
                MoveRight(p,k);
            else
                if (p->branch[k+1]->count > MIN)
                    MoveLeft(p,k+1);
                else
                    Combine(p,k);
}

void Remove(Node_type *p, int k)
{
    int i;
    for (i=k+1; i<=p->count; i++){
        p->key[i-1] = p->key[i];
        p->branch[i-1] = p->branch[i];
    }
    p->count--;
}

void Successor(Node_type *p, int k)
{
    Node_type *q;
    for (q=p->branch[k]; q->branch[0]; q=q->branch[0])
    ;
    p->key[k]=q->key[1];
}

void PushIn(Key_type x, Node_type *xr, Node_type *p, int k)
{
    int i;
    for (i = p->count; i > k; i--) {
        p->key[i+1] = p->key[i];
```
```c
p->branch[i+1] = p->branch[i];
}
p->key[k+1] = x;
p->branch[k+1] = xr;
p->count++;
}

// Convert #10
// How can parameters be call-by-ref in Java? Not allowed to make global.
// Try to think of a simple way to handle all of these call-by-ref params
// in the source - without making a mistake or the program will be
// impossible to debug.

void PrintNode(Node_type *p, int depth, int i, int *pos) // CALL-BY-REF
{
    int j;
    while ((*pos)<loc[depth][i]) {
        printf(" ");
        (*pos)++;
    }
    printf("[");
    for (j=1; j<p->count; j++){
        printf("%c",p->key[j]);
        (*pos) += 2;
    }
    printf("%c",p->key[j]);
    (*pos)+=3;
}

// Convert #11
void PrintTop(Node_type *p, int depth, int i, int *pos, int *child) // CALL-BY-REF
{
    int j,strt,mid;
    mid = loc[depth][i]+p->count;
    for (j=0; j<=p->count; j++) {
        strt = loc[depth+1][*child]+1;
        if (strt>=mid)
            strt += 2*p->branch[j]->count-2;
        while ((*pos)<strt) {
            printf(" ");
            (*pos)++;
        }
        if (strt<=mid)
            printf("/");
        else
            printf("\\");
        (*pos)++;
        (*child)++;
    }
}
```
void PrintLine(Node_type *p, int depth, int i, int *pos, int *child) // CALL-BY-REF
{
    int strt, stop, mid;

    strt = loc[depth+1][*child]+2;
    stop = loc[depth+1][*child+p->count]+p->branch[p->count]->count*2-2;
    mid = loc[depth][i]+p->count;
    *child += p->count+1;
    while ((*pos)<strt) {
        printf(" ");
        (*pos)++;
    }
    while ((*pos)<=stop) {
        if (*pos==mid)
            printf("|");
        else
            printf("_`");
        (*pos)++;
    }
}

void PrintAllNodes(Node_type *root) {
    int i, depth, pos, child;

    if (root==NULL)
        printf("EMPTY
");
    else {
        for (depth=0; depth<MAXD; depth++)
            cnt[depth]=0;
        levels=0;
        DFS(root,0);
        SetLoc();
        printf("L
");
        for (depth=0; depth<=levels; depth++) {
            printf("%d",depth);
            pos=-1;
            for (i=0; i<cnt[depth]; i++)
                PrintNode(ptrs[depth][i],depth,i,&pos);
            printf("\n");
            if (depth < levels) {
                pos=-2;
                child=0;
                for (i=0; i<cnt[depth]; i++)
                    PrintLine(ptrs[depth][i],depth,i,&pos,&child);
                printf("\n");
                pos=-2;
                child=0;
            }
        }
    }
}
for (i=0; i<cnt[depth]; i++)
    PrintTop(ptrs[depth][i], depth, i, &pos, &child);
    printf("\n");
}
printf("\n");
}

// Convert #14
Bool SeqSearch(Key_type target, Node_type *p, int *k) // CALL-BY-REF
{
    if (target < p->key[1]) {
        *k = 0;
        return FALSE;
    }
    else {
        *k = p->count;
        while ((target<p->key[*k]) && *k > 1) {
            (*k)--;  
            step++;
        }
        return (target==p->key[*k]);
    }
}

// Convert #15
void Split(Key_type x, Node_type *xr, Node_type *p, int k, Key_type *y, Node_type **yr) // CALL-BY-REF
{
    int i;
    int median;

    if (k <= MIN)
        median = MIN;
    else
        median = MIN + 1;
    *yr = (Node_type *)malloc(sizeof(Node_type));
    for (i = median+1; i <= MAX; i++) {
        (*yr)->key[i-median] = p->key[i];
        (*yr)->branch[i-median] = p->branch[i];
    }
    (*yr)->count = MAX - median;
    p->count = median;
    if (k <= MIN)
        PushIn(x, xr, p, k);
    else
        PushIn(x, xr, *yr, k - median);
    *y = p->key[p->count];
    (*yr)->branch[0] = p->branch[p->count];
    p->count--;
// Convert #16
Bool PushDown(Key_type newkey, Node_type *p, Key_type *x, Node_type **xr) // CALL-BY-REF
{
    int k;
    if (p == NULL) {
        *x = newkey;
        *xr = NULL;
        return TRUE;
    }
    else {
        if (SeqSearch(newkey, p, &k))
            Error("inserting duplicate key");
        if (PushDown(newkey, p->branch[k], x, xr))
            if (p->count < MAX) {
                PushIn(*x, *xr, p, k);
                return FALSE;
            }
        else {
            Split(*x, *xr, p, k, x, xr);
            return TRUE;
        }
        return FALSE;
    }
}

// Convert #17
Bool RecDelete(Key_type target, Node_type *p)
{
    int k;
    Bool found;

    if (p == NULL)
        return FALSE;
    else {
        found = SeqSearch(target, p, &k);
        if (found)
            if (p->branch[k-1]) {
                Successor(p, k);
                if (!found == RecDelete(p->key[k], p->branch[k]))) Error("Key not found.");
            } else
                Remove(p, k);
        else
            found = RecDelete(target, p->branch[k]);
        if (p->branch[k] != NULL)
            if (p->branch[k]->count < MIN)
                Restore(p, k);
        return found;
    }
}
/ C version
Node_type *Insert(Key_type newkey, Node_type *root)

{ /* NEED TO CONVERT THE BODY */
  Key_type x;
  Node_type *xr;
  Node_type *p;
  Bool pushup;

  pushup = PushDown(newkey, root, &x, &xr);
  if (pushup) {
    p = (Node_type *)malloc(sizeof(Node_type));
    p->count = 1;
    p->key[1] = x;
    p->branch[0] = root;
    p->branch[1] = xr;
    return p;
  }
  return root;
}

} // end Btree class
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
● **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();
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:

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

(new ConcreteProgram1()).start();

// Awkward solution (not easily extended to other classes):

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

switch (code) {
    case 1: (new ConcreteProgram1()).start();
        break;
    case 2: (new ConcreteProgram2()).start();
        break;
}
```
• 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
Abstract Factory Pattern: Platform Independent GUI

- **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
#elif PLATFORM == MS
    // put MS code here
#elif 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

```
main(args: String[]) {
  Integer code = parseInt(args[0]);
  AbstractFactory f = getFactory(code);
  Menu m = createMenu(f);
  Button b = createButton(f);
  select(m);
  push(b);
}
```

**Abstract Factory**
- `SunFactory`
  - `createMenu(): Menu`
  - `createButton(): Button`
- `MSFactory`
  - `createMenu(): Menu`
  - `createButton(): Button`

**GUI**
- `main(args: String[])`
  - `Menu select()`
  - `SunMenu select()`
  - `MSMenu select()`
  - `Button push()`
  - `SunButton push()`
  - `MSButton push()`

- `System.out.println("SBP")`
- `System.out.println("SMS")`
- `Integer code = parseInt(args[0]);`
- `AbstractFactory f = getFactory(code);`
- `Menu m = createMenu(f);`
- `Button b = createButton(f);`
- `select(m);`
- `push(b);`

- 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
- 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:

```plaintext
Supplier #: S1
Name: Smith
Status: 20
City: London

SUBMIT
```

```plaintext
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:

```plaintext
Supplier #: S1
Name: Smith
Status: 30
City: Paris

SUBMIT
```

```plaintext
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.

![Diagram of Application constructing PersistentObject](image)

- 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:

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

- 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.

![Diagram of Remote Proxy Pattern]

- 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.
• 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
/ 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 AssertionFailedError(s + ":" + condition);
    }
    public static void assertEquals(int expected, int actual) {
        if (expected != actual)
            throw new AssertionFailedError(expected + "<>" + actual);
    }
    public static void assertEquals(String expected, String actual) {
        if (!expected.equals(actual))
            throw new AssertionFailedError(expected + "<>" + actual);
    }
}

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

// All TestCases and TestSuites implement Test, hence a run() method, containing a log
public interface Test {
    public void run(TestResult 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(TestResult result) {
        for (Enumeration e = fTests.elements(); e.hasMoreElements();) {
            Test test = (Test)e.nextElement();
            test.run(result);
        }
    }
    public void addTest(Test test) {
        fTests.addElement(test);
    }
}
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();
    }
}

public class TestResult {
    protected Vector fErrors = new Vector();
    protected Vector fFailures = new Vector();
    public synchronized voidaddError(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<fErrors.size(); i++) {
            TestFailure testFailure = (TestFailure)fErrors.elementAt(i);
            System.out.println(testFailure);
        }
        System.out.println("Failures:");
        for (int i=0; i<fFailures.size(); i++) {
            TestFailure testFailure = (TestFailure)fFailures.elementAt(i);
            System.out.println(testFailure);
        }
    }
}
JUnit: Framework Java Code

// 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 AssertFailedErrors 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);
    }
}
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
• 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 8) 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.sqlite.JDBC"

private String url = "jdbc:mysql:../sqlite/"

• DataBase is constructed by the Supplier when it is constructed
• Constructor for DataBase initializes private variables
• Base URL indicates the directory where files (DB) are stored
• URL needs to have the database name appended to the base URL
• Note the location of the drivers (these will be found in the given JDBC jar file)
DataBase: Query

- 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 locates files (DB)
  - 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()) { // 6th line of JDBC
      System.out.println(rs.getString("S_NO")); // 7th line of JDBC
      rs.close(); // 8th 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

- 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)
- close() ends the READ action, allows READ or WRITE action later (8th line of JDBC)
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

- 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

Supplier supp

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: login to linux
//    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:"
        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
Remote/Persistent Objects: Supplier Exercise

- 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:

  ms% start rmiregistry
  ms% start java -Djava.security.policy=java.policy SupplierServer
  ms% java -Djava.security.policy=java.policy SupplierClient