```
  public void setBackEnd(BackEndInterface be)
  { b = be; }
  ...
}

public interface BackEndInterface
{ ...
  signatures and specifications of methods
  of back end used by Gui
}

public class BackEnd implements BackEndInterface
{ ... }
```

This allows any class that implements *BackEndInterface* to be used by *Gui* class.

Part 3: Software Architectures

Architecture of piece of software is overall organisation of program code into separate subparts, responsible for different aspects of functionality.

Eg, many traditional application programs can be separated into a 'Front end' responsible for dealing with interaction with users (ie, the GUI), and a 'Back end' responsible for storing data and performing calculations in response to user requests (relayed via GUI).

Many web-based applications can similarly be separated into a client-side GUI (HTML pages and forms) and server-side processing (data access/computations and generation of web pages).

Benefit is that these separate parts can be developed, tested and modified independently, and may be deployed on separate hardware.

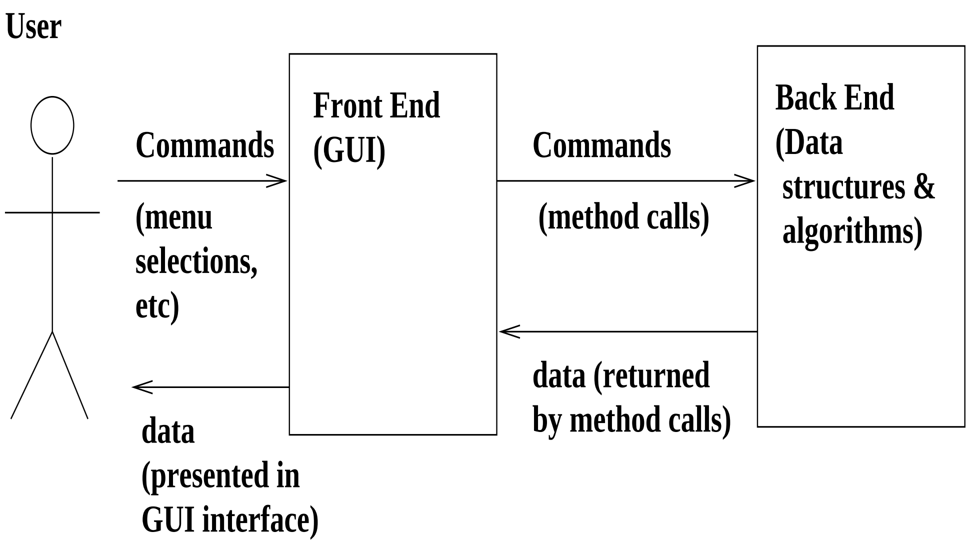
Front End/Back End Architectures

Also called 'two tier' or client/server architectures and define 2 separate subsystems/tiers: GUI and rest of application.

Example of this architecture is address book application (from CS1PR2) supporting storage of (name, address, phone no.) records, which has menu-based Java Swing interface providing options to enter records and search for records with particular values for name and other fields.

Back end of system consists of classes *DataItem* representing individual records as objects, and *Database*, containing list of these records. Front end consists of *JFrame* with menus for commands such as *Add* and *Search*, and listener methods to detect and act on user selections of these menus. Also presents results of commands (eg, in a *JTextArea* or *JList*).

Typical Front End/Back End Architecture



DataItem could be defined by:

```
public class DataItem   /* Holds individual entries */
{ private String name = "";
  private String address = "";
  private String phone = "";

  public DataItem(String nme, String addr, String ph)
  { name = nme;
    address = addr;
    phone = ph;
  }

  public String getName()
  { return name; }

  public String getAddress()
  { return address; }
```



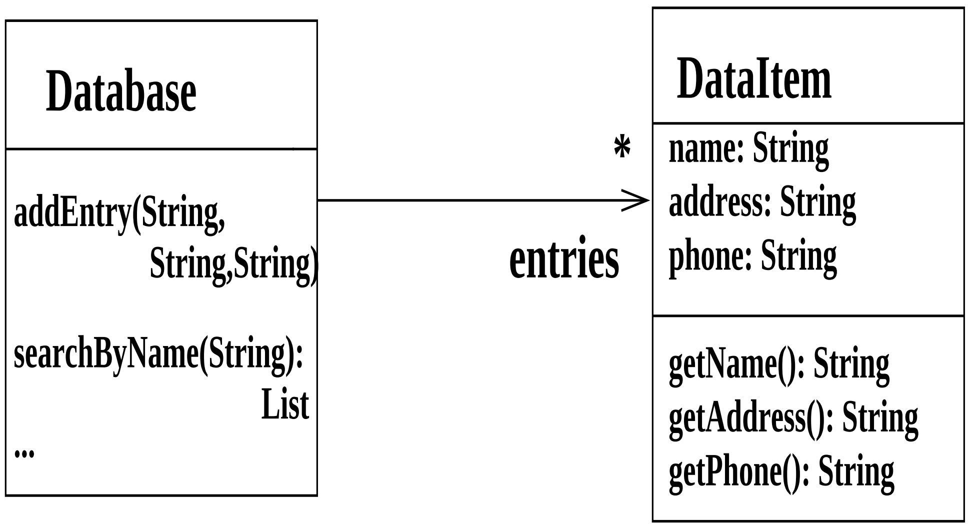
```
  public String getPhone()
  { return phone; }

  public String toString()
  { return name + " @ " + address + " Ph: " + phone; }
}
```

The *Database* class which encapsulates stored data is:

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

public class Database
{ private List entries = new ArrayList();  /* of DataItem */

  public void addEntry(String nme, String addr, String ph)
  { DataItem di = new DataItem(nme,addr,ph);
    entries.add(di);
  }
```

```
  public List searchByName(String nme)
  { List res = new ArrayList();
    for (int i = 0; i < entries.size(); i++)
    { DataItem di = (DataItem) entries.get(i);
      if (di.getName().equals(nme))  /* Exact match */
      { res.add(di); }
    }
    return res;
  }

  /* Similarly for searchByAddress, etc */
}
```


UML Diagram of Back End



This is complete back end. Can be tested before a GUI is written, by putting test scenarios in the *main* method of *Database*:

```
public static void main(String[] args)
{ Database db = new Database();
  db.addEntry("Alex Ferguson",
      "MUFC, Old Trafford, Manchester",
      "+44 (0)161 000 2343");
  db.addEntry("Ann Widdecombe",
      "House of Commons, Westminster", "");
  List res = db.searchByName("Alex Ferguson");
  System.out.println(res);
}
```


Database class is only interface between front and back ends: the GUI calls $database.searchByName(n)$ with string parameter n entered by user, and gets back a list of matching data items.

GUI and back end can be modified relatively independently: could change back end to use *Map* or *Vector* data structures instead of *ArrayList*, or to interface to an Access database, etc, without any change to GUI.

Likewise can change GUI to use a ‘direct manipulation’ style of interaction where user flicks through a visual simulation of address book instead of using menus.

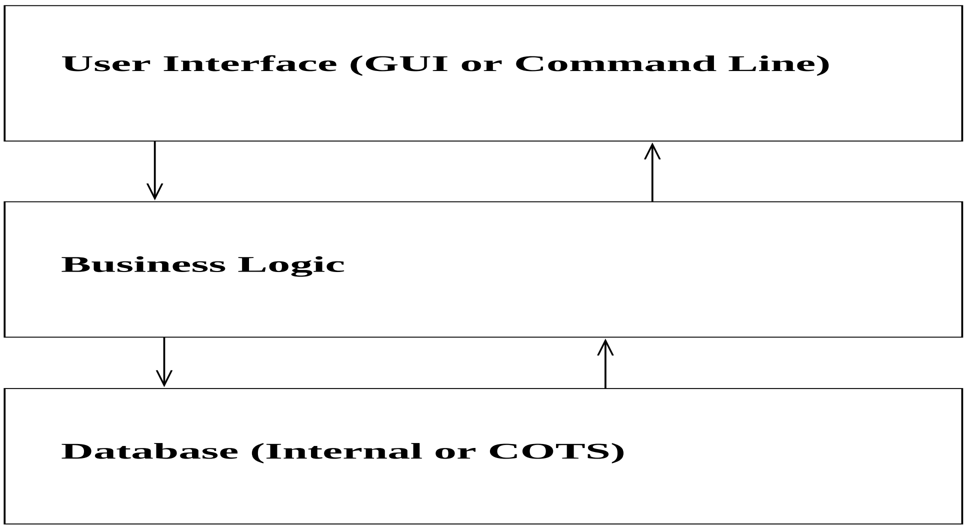
Three Tier Architectures

Many commercial applications use a *Three Tier Architecture*, in which separate levels are: (1) GUI; (2) Business logic; (3) Data repository.

This separates subparts of program into separate areas of responsibility: GUI deals with user interaction, business logic performs computations, and data repository stores persistent data (ie, data is stored permanently from one execution of program to another).

Any of the tiers may use pre-existing code or external applications, commercial off-the-shelf products (COTS), etc to carry out all or part of their functionality, this will be usual case with database tier in particular.

Three Tier Architecture



Design of Databases using JDBC

JDBC (Java database connectivity) provides way to connect Java programs to databases in Access, PostgreSQL, Oracle, and many other database systems.

It allows programs to make queries on and modify database tables using standard SQL (Structured Query Language) statements.

We will turn the address book application into a three-tier application using JDBC.

Reference: http://www.java.sun.com/jdbc

Address Book Application using Access

First step is to design database tables for entities to be stored: the *AddressBookEntry* entity.

Identify primary key *AddressBookEntryID* and other columns of table for this entity (*Name*, *Address* and *Phone* all of type *String*).

Construct table in blank database, using Access. Select (32 bit) ODBC from Control Panel in Windows. Open ODBC icon, and in "System DSN" menu, choose Access driver, and option "Add" to select physical database file and name and location for ODBC interface to it: this name is "data source name" (DSN) *AddressDb* which will be used in database URL in Java program. An appropriate location for DSN is ODBC/Data Sources. Repeat process for "File DSN" menu.

Here is an example of data that could be stored in *AddressBookEntry* table:

AddressBook EntryID	*Name*	*Address*	*Phone*
1	J. Fox	123 Houndsditch	0787 909 112
2	A. Widdecombe	Smith Square	
3	A. Ferguson	Old Trafford	club call 909
4	A. Ferguson	1087 Salford Quays	

In a Java program, we need to load JDBC-ODBC bridge class:

```
import java.sql.*;

public class AddressBookDb
{ public static void main(String args[])
  { try
    { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
    catch(Exception e)
    { System.out.println(e.toString());
      System.exit(1);
    }
    ...
```

Once driver class is loaded, the program can attempt to connect to the database:

```
try
{ String s = "jdbc:odbc:AddressDb";  /* URL of database */
  Connection con = DriverManager.getConnection(s);
  System.out.println("Connection established");
  ... <process database> ...
}
catch(SQLException sqle)
{ while (sqle != null)
  { System.out.println("SQLState: " + sqle.getSQLState());
    System.out.println("Message: " + sqle.getMessage());
    System.out.println("ErrorCode: " + sqle.getErrorCode());
    sqle = sqle.getNextException();
    System.out.println(" ");
  }
}
```


There are four basic operations on a database, expressed in SQL:

1. Adding a new item to a database table: `INSERT INTO table VALUES (keyval,val1,val2,...,valn)`
2. Searching for a database item: `SELECT * FROM table WHERE att = val`
3. Deleting a database item: `DELETE * FROM table WHERE att = val`
4. Updating a database item: `UPDATE table SET att = val WHERE P`

Adding an Item to a Table

If *con* is database *Connection*, create *INSERT* statement and execute it by:

```
Statement stat = con.createStatement();
String insert = "INSERT INTO AddressBookEntry " +
                "VALUES (" + entryID + ",'" + name + "','" +
                         address + "','" + phone + "')";
stat.executeUpdate(insert);
con.commit();
```

inside `try` with `catch(SQLException sqle)`. *entryID* is an integer variable, *name*, *address*, *phone* are strings. Table name and attributes must be specified correctly, otherwise exception occurs. Any statement which modifies database needs a *commit* following it to confirm changes.

Java String values should be enclosed in single quotes for correct translation to database string values.

Searching for an Item in a Table

If *con* is database *Connection*, *nameval* holds name to search for, create *SELECT* statement and execute it by:

```
Statement stat = con.createStatement();
String find = "SELECT * FROM AddressBookEntry " +
              "WHERE Name + " = '" + nameval + "'";
ResultSet res = stat.executeQuery(find);
```

Eg, to get all items in the table, do:

```
public void displayAll()
{ String sQuery = "SELECT * FROM AddressBookEntry";
  ResultSet res;
  try
  { Statement stat = con.createStatement();
    res = stat.executeQuery(sQuery);
  }
}
```

Processing Query Results

A list *res* is returned from queries. Need to iterate to obtain individual results (the table rows that satisfy query):

```
while (res.next())
{ int sId = res.getInt("AddressBookEntryID");
  String sName = res.getString("Name");
  String sAddress = res.getString("Address");
  String sPhone = res.getString("Phone");
  System.out.println("Entry number " + sId + " Name: " + sName +
                  " Address: " + sAddress + " Phone: " + sPhone);
}

res.close();
```

All results from query are accessed by `res` iterator. Individual fields of result row are accessed by *getType* operators: Type can be Int, Float, Double, String, Boolean, etc.

Deleting Items

Let *con* be established database *Connection* object. Create a *DELETE* statement and execute it by:

```
Statement stat = con.createStatement();
String del = "DELETE * FROM AddressBookEntry " +
  "WHERE name + " = '" + nameVal + "'";
stat.executeUpdate(del);
con.commit();
```

with these statements inside a `try` with a `catch(SQLException sqle)`.

Sun documentation for JDBC is at `http://java.sun.com/jdbc/` including information on prepared statements and stored procedures, which have improved efficiency over basic techniques given above.

Complete Address Book Database Interface

Rewritten database interface for address book application, replacing *Database* class, is then:

```
import java.sql.*;
import java.util.Vector;
import java.util.List;

public class AddressBookDb
{ private Connection con;  // Connection to address book database
  private int entryId = 0; // Max. id of entries

  public AddressBookDb(String dbName)
  { try
    { String s = "jdbc:odbc:" + dbName;
      con = DriverManager.getConnection(s);
      System.out.println("Connection established");
      entryId = getMaxId();
```



```
    }
    catch(SQLException sqle)
    { System.err.println("Unable to connect to " + dbName);
      handleException(sqle);
    }
  }

  private void handleException(SQLException e)
  { e.printStackTrace();
    while (e != null)
    { System.out.println("SQLState: " + e.getSQLState());
      System.out.println("Message: " + e.getMessage());
      System.out.println("Code: " + e.getErrorCode());
      e.getNextException();
    }
  }

  private int getMaxId()  // Returns max primary key
```

```
{ int maxId = 0;        // value used in database.
  try
  { Statement stat = con.createStatement();
    String sQuery = "SELECT MAX(AddressBookEntryID) " +
                    "AS MaxId FROM AddressBookEntry";
    ResultSet res = stat.executeQuery(sQuery);
    while (res.next())
    { maxId = res.getInt("MaxId"); }
  }
  catch(SQLException e)
  { System.err.println("Error in getting max id");
    handleException(e);
  }
  return maxId;
}

public void addEntry(String nme, String addr, String ph)
{ Statement stat;
```



```
  entryId = getMaxId(); // To ensure unique id's are used
  String insert = "INSERT INTO AddressBookEntry " +
    "VALUES (" + (entryId+1) + ",'" + nme + "','" +
              addr + "','" + ph + "')";
  try
  { stat = con.createStatement();
    stat.executeUpdate(insert);
    entryId++;
    con.commit();
  }
  catch(SQLException sqle)
  { System.err.println("Adding an entry failed");
    handleException(sqle);
  }
}

public void displayAll()
{ String sQuery = "SELECT * FROM AddressBookEntry";
```

```
  ResultSet res;
  try
  { Statement stat = con.createStatement();
    res = stat.executeQuery(sQuery);
    while (res.next())
    { int sId = res.getInt("AddressBookEntryID");
      String sName = res.getString("Name");
      String sAddress = res.getString("Address");
      String sPhone = res.getString("Phone");
      System.out.println("Entry number " + sId +
                         "\t Name: " + sName +
                         "\t Address: " + sAddress +
                         "\t Phone: " + sPhone);
    }
    res.close();
  }
  catch(SQLException e)
  { System.err.println("Display-all failed");
```



```
    handleException(e);
  }
}

public List searchByName(String nameval)
{ Statement stat;
  ResultSet res;
  Vector items = new Vector();
  String find = "SELECT * FROM AddressBookEntry " +
                "WHERE Name = '" + nameval + "'";
  try
  { stat = con.createStatement();
    res = stat.executeQuery(find);

    while (res.next())
    { int sId = res.getInt("AddressBookEntryID");
      String sName = res.getString("Name");
      String sAddress = res.getString("Address");
```

```
      String sPhone = res.getString("Phone");
      System.out.println("Entry number " + sId +
                         "\t Name: " + sName +
                         "\t Address: " + sAddress +
                         "\t Phone: " + sPhone);
      DataItem item = new DataItem(sName,sAddress,sPhone);
      items.add(item);
    }
    res.close();
  }
  catch(SQLException e)
  { System.err.println("Search by name failed");
    handleException(e);
    return items;
  }
  return items;
}
```



```
public void remove(String nameVal)
{ Statement stat;
  String del = "DELETE * FROM AddressBookEntry " +
               "WHERE Name = '" + nameVal + "'";
  try
  { stat = con.createStatement();
    stat.executeUpdate(del);
    con.commit();
  }
  catch(SQLException e)
  { System.err.println("Deletion failed");
    handleException(e);
  }
}

public static void main(String args[])
{ try
  { Class.forName("sun.jdbc.odbc.JdbcOdbcDriver"); }
```

```
    catch(Exception e)
    { System.out.println(e.toString());
      System.exit(1);
    }
    AddressBookDb dbInterface =
      new AddressBookDb("AddressDb");
    dbInterface.addEntry("T. Fox", "123 Houndsditch",
                         "0787 909 112");
    dbInterface.displayAll();
    dbInterface.addEntry("A. Widdecombe", "Smith Square", " ");
    dbInterface.addEntry("A. Ferguson", "Old Trafford",
                         "club call 909");
    dbInterface.addEntry("A. Ferguson","1087 Salford Quays"," ");
    dbInterface.searchByName("A. Ferguson");
    dbInterface.remove("A. Ferguson");
    dbInterface.displayAll();
  }
}
```


Result of executing the above test in *main* is:

```
WIN J:\teaching\cs2sap\Code2>java AddressBookDb
Connection established
Entry number 1 Name: A Jennings
               Address: 123 Houndsditch
               Phone: 33445 44-55
Entry number 2 Name: T. Fox
               Address: 123 Houndsditch
               Phone: 0787 909 112
Entry number 4 Name: A. Ferguson
               Address: Old Trafford
               Phone: club call 909
Entry number 5 Name: A. Ferguson
               Address: 1087 Salford Quays
               Phone:
Entry number 1 Name: A Jennings
               Address: 123 Houndsditch
```

```
                Phone: 33445 44-55
Entry number 2 Name: T. Fox
                Address: 123 Houndsditch
                Phone: 0787 909 112
Entry number 3 Name: A. Widdecombe
                Address: Smith Square
                Phone:
```

where database already contained the entry for *A Jennings*.

The *DataItem* class is still used to transfer query results from database interface to GUI, but *Database* class has been entirely replaced by *AddressBookDb*.

Part 4: Design Patterns

Design patterns are standard design structures which have been found to be useful in many different applications. They provide a set of ready-made solutions to design problems which are encountered in software development, and a repertoire of building blocks for software design.

Design patterns are an important technique for organising structure of components (modules or subsystems) in a clear and systematic fashion.

Three main categories: *Creational* patterns address object construction; *Structural* patterns focus on static structure of inheritance and clientship, *Dynamic* patterns deal with object behaviour.