Practical Session Structure

1. Introduction
2. Building a business component
3. Building an admin GUI
4. Introducing .NET remoting
5. Creating a web service and client website
6. Developing a Java client

Overview

• In this lecture we will build a business component which encapsulates the business functionality of the system

In this practical session i will:

– Briefly explain the n-tier architecture and components and see how they could be used in a distributed system

– Build a business component which encapsulates the core functionality of the system

N-Tier Architectures

• In N-Tier architectures there is a logical separation of presentation, business and data into separate layers

• Data Tier – manages the data

– The database we built last week

• Presentation Tier – controls what a user sees and can do with the system

– We will build several applications within this tier later on

• Business Tier (middle tier) – controls everything else (the business logic)

– What we will build today

Business Tier

• The business tier contains the core functionality of our system

– Business rules

– Work flows

• It provides controlled access to data

• It enables validation and processing of data input

Components

• Our component will consist of a collection of classes developed to fulfil a certain specification

• It can be re-used

• It should encapsulate all its behaviour

• It must provide an interface to allow it to be accessed by a client (could even be another component)

Getting Started

• We will create our business component as a Class Library in Visual Studio 2005

• Create a project

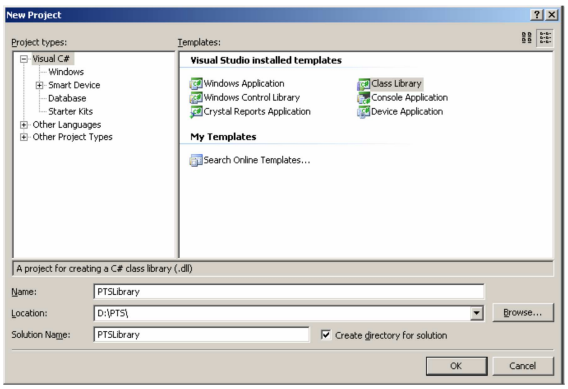
– Open Microsoft Visual Studio 2005

– Go to File -> New Project

– Select Visual C# as the project type and then select Class Library as the template

– Name the project PTSLibrary and save it in a suitable location (PTS = Project Tracking System)

Creating the Project



PTSLibrary Structure

• As a business component, this project will not contain any graphical user interface

• There are three types of classes we will have in our project:

– Business Objects

– DAOs (Data Access Objects)

– Façade Objects

Business Objects

• Business objects (also called domain objects) are abstract representations of entities from our business domain
• They represent concepts that are important to the business that the system is modelling
• In our system these are abstractions of project management related concepts, such as project, team, task, etc.

The business objects in our component will be:
– Project
– Task
– Subtask
– User
– Team
– TeamLeader
– TeamMember
– Customer
– Status

Some of the business objects have the same name as entities in our data model, but not all. There are business objects not in the data model
• Relational data models require a different approach than object-oriented modelling
– Object-oriented paradigm is based on software engineering principles
– Relational paradigm is based on mathematical principles
• Working with the two models can lead to problems referred to as "Object-Relational Impedance Mismatch"

Data Access Objects provide abstract interfaces to data sources
• DAOs provide a clear separation between our business and persistence logic
• We want to write robust code and achieve low coupling between our business classes and the database
– No need to clutter our business logic with SQL code

– No need to rewrite all our business classes if there is a change in the database

The DAOs will contain all the SQL code for reading and writing to the database

• There could be one DAO for the entire project, but as we have different types of user, working with different data, we will have a DAO for each role:

– SuperDAO (super class for all others)

– AdminDAO

– CustomerDAO

– ClientDAO (team leaders using Java or .NET clients)

Façade Objects

• The PTSLibrary project is a class library

– No graphical user interface

– Used by other sections of our system, which shouldn't know about the inner structure of our business component

• We provide a publicly available interface to our business component via façade classes

Again we will have one façade class for each type of user who will access our business component

– This also allows us to show each role of user only what they need to see (e.g. we wouldn't want a team leader to be able to create a new project, only administrators)

• The façade classes are:

– PTSAdminFacade

– PTSClientFacade

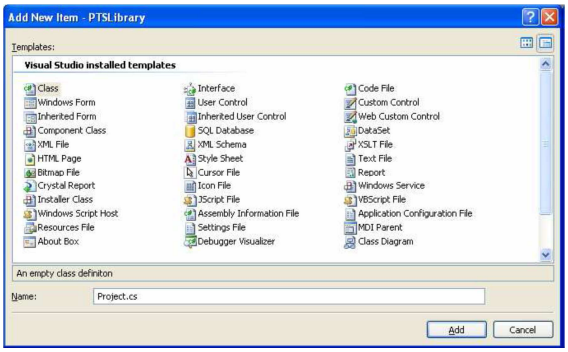
– PTSCustomerFacade

– PTSSuperFacade (super class for all others)

Delete the Class1.cs file created by default when you created the new project

• Now create all the business classes

• Make sure that you select Class as the template for each



You Solution Explorer should now look like this

• Each of the classes created only contains some default import statements (using statements), namespace declaration and class declaration. Let's add our desired state and behavior

• Remember that we will only implement a subset of the functionality required to demonstrate the use of the system

Class User

• This class represents a general user of the system

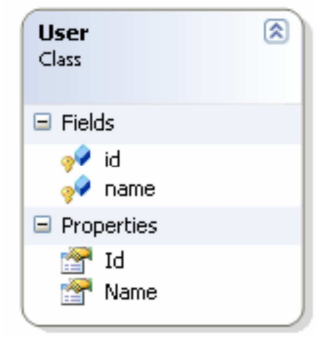
• It is the super (base) class for all more specialised classes representing users

– Customer

– TeamLeader

– TeamMember

• The above three inherit from User



This class has only two protected variables (username and password), which are exposed through two read-only properties

• Note that the access level is set to protected

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PTSLibrary
{
    class User
    {
        protected string name;
        protected int id;

        public string Name
        {
            get { return name; }
        }

        public int Id
        {
            get { return id; }
        }
    }
}
```

Class Customer

• This class represents a customer (someone who commissioned a project)

• The functionality we want to provide for this class is simplified

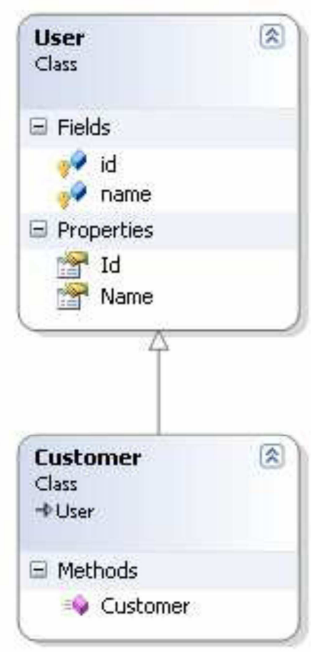
– Keep the name and show it when required

– This functionality already exists in the User class, so we make Customer inherit from it

• We also want instances of this class with the name set, so we need to create a constructor to allow us to do this

• All the code we need to write is – make the Customer inherit from User

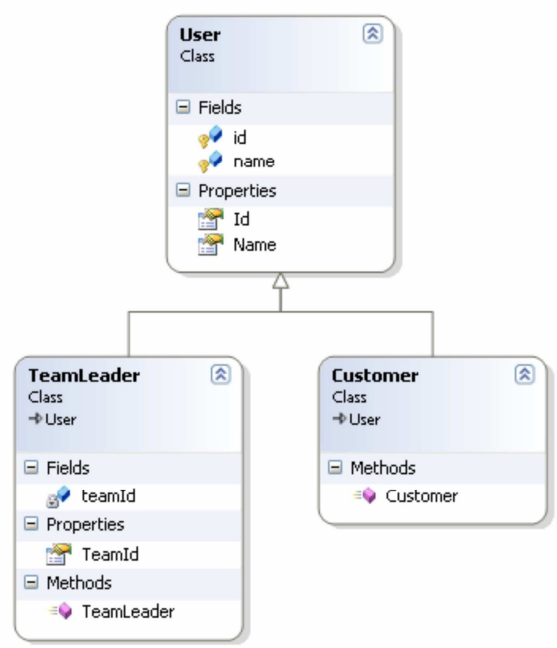
– add a constructor taking a name and id



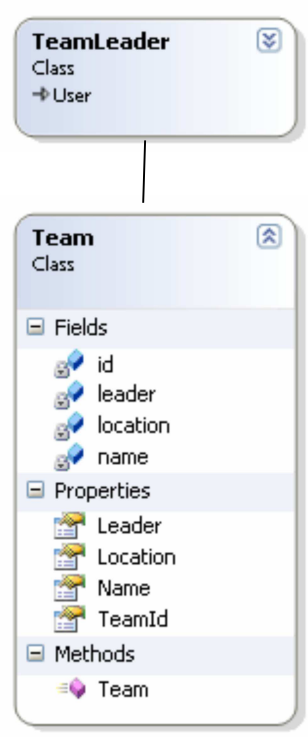

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PTSLibrary
{
    class Customer : User
    {
        public Customer(string name, int id)
        {
            this.name = name;
            this.id = id;
        }
    }
}
```

A: the name and id members are not declared in Customer, but are inherited from User (This is some of the code that was missing in what I had sent initially)

Class TeamLeader




```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace PTSLibrary
6  {
7      class TeamLeader : User
8      {
9          private int teamId;
10
11         public int TeamId
12         {
13             get { return teamId; }
14             set { teamId = value; }
15         }
16
17         public TeamLeader(string name, int id, int teamId)
18         {
19             this.name = name;
20             this.id = id;
21             this.teamId = teamId;
22         }
23     }
24 }
```


TeamLeader
Class
User
Team
Class
Fields
id
leader
location
name
Properties
Leader
Location
Name
TeamId
Methods
Team

```
7   class Team
8   {
9       private int id;
10      private string location;
11      private string name;
12      private TeamLeader leader;
13
14      public int TeamId
15      {
16          get { return id; }
17          set { id = value; }
18      }
19
20      public TeamLeader Leader
21      {
22          get { return leader; }
23          set { leader = value; }
24      }
25
26      public string Location
27      {
28          get { return location; }
29          set { location = value; }
30      }
31
32      public string Name
33      {
34          get { return name; }
35          set { name = value; }
36      }
37
38      public Team(int id, string location, string name, TeamLeader leader)
39      {
40          this.location = location;
41          this.name = name;
42          this.id = id;
43          this.leader = leader;
44      }
45  }
```

Enum Status

• Enum allows you to create a distinct value type

• Contains a set of named constants

• Can be converted to an integer

• Which is easier to read?

– if(currentStatus == 3)

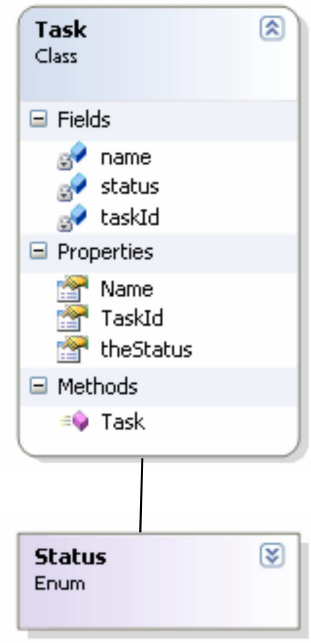
– If(currentStatus == Status.Completed)

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace PTSLibrary
6  {
7      public enum Status
8      {
9          ReadyToStart = 1,
10         InProgress = 2,
11         Completed = 3,
12         WaitingForPredecessor = 4
13     }
14 }
15
```

States of a task or subtask

• Note the change from class to enum

• The integer numbers assigned to each status reflect the StatusId field in the Status table of the database



• Represents a task within a project, which is assigned to a team and can be broken into subtasks

• Linked to Status through association

• Other fields: name and taskId

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PTSLibrary
{
    class Task
    {
        private Guid taskId;
        private string name;
        private Status status;

        public Guid TaskId
        {
            get { return taskId; }
            set { taskId = value; }
        }

        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        public Status theStatus
        {
            get { return status; }
            set { status = value; }
        }

        public Task(Guid id, string name, Status status)
        {
            this.taskId = id;
            this.name = name;
            this.status = status;
        }
    }
}
```

• Notice the use of the Guid data type

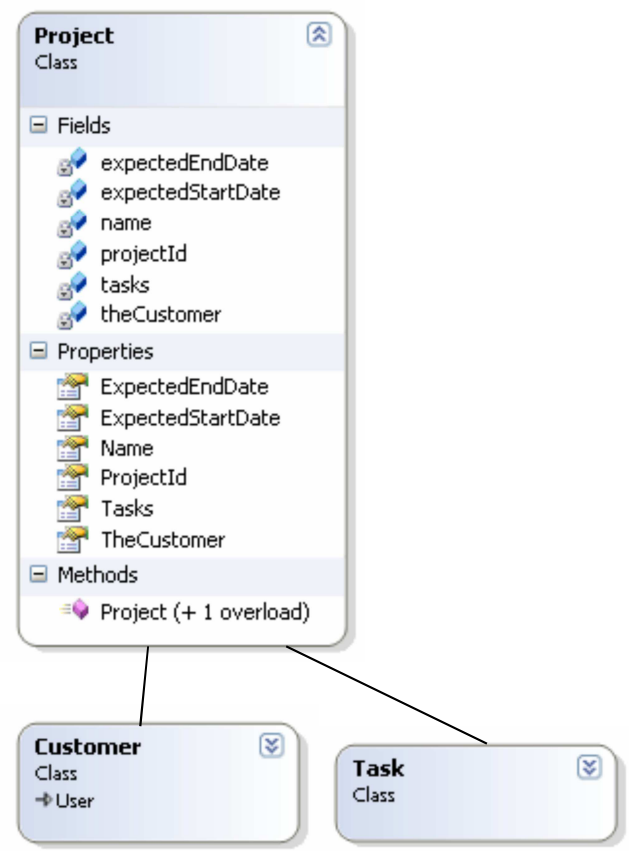
Class Project

• Represents a project

• Linked to Customer and Task through association

• Has two constructors which set different fields

– Depending on the context in which the Project object is used



• Note the data types used in declaring the variables

• tasks is declared using generic programming

• Generics:

– Allow the creation of typesafe collections

– tasks is a list that can contain objects of type Task only

– Declared using <Type>

```
private string name;
private DateTime expectedStartDate;
private DateTime expectedEndDate;
private Customer theCustomer;
private Guid projectId;
private List<Task> tasks;
```

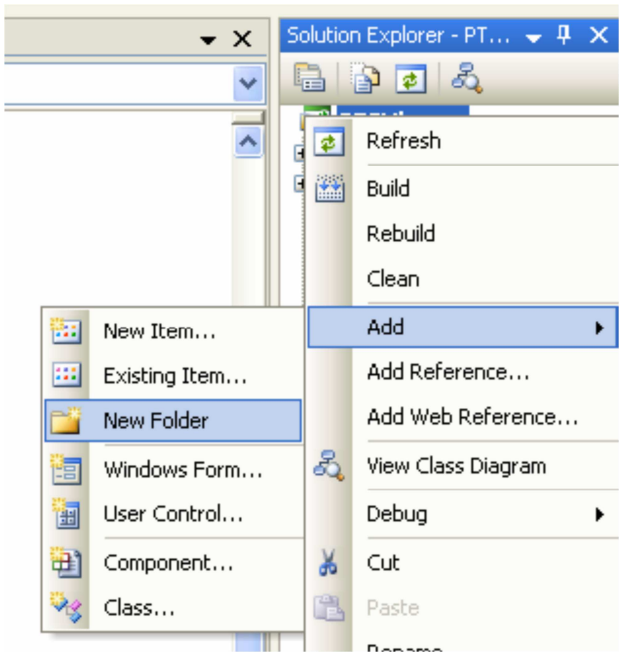
• Two constructors (one setting the customer, one the tasks)

```
public Project(string name, DateTime startDate, DateTime endDate, Guid projectId, Customer customer
{
    this.name = name;
    this.expectedStartDate = startDate;
    this.expectedEndDate = endDate;
    this.projectId = projectId;
    this.theCustomer = customer;
}

public Project(string name, DateTime startDate, DateTime endDate, Guid projectId, List<Task> tasks)
{
    this.name = name;
    this.expectedStartDate = startDate;
    this.expectedEndDate = endDate;
    this.projectId = projectId;
    this.tasks = tasks;
```

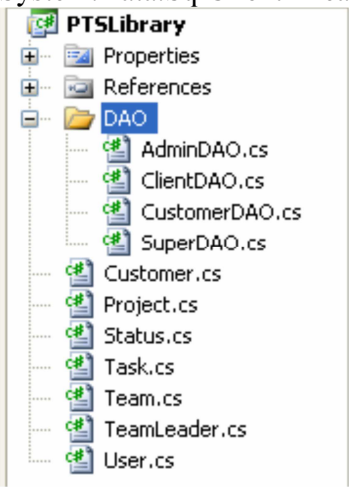
Creating DAOs

• We will keep all the DAOs in a subfolder of our project to have all DAOs in one place

• Create a new folder in the PTSLibrary project called DAO



• Create 4 new classes in the DAO folder called:
– SuperDAO
– AdminDAO
– CustomerDAO
– ClientDAO
• These classes will have code to work with our database
– To have access to the required classes we need to import namespaces System.Data and System.Data.SqlClient in each DAO class




```
using System;
using System.Collections.Generic;
using System.Text;
using System.Data.SqlClient;
using System.Data;
```

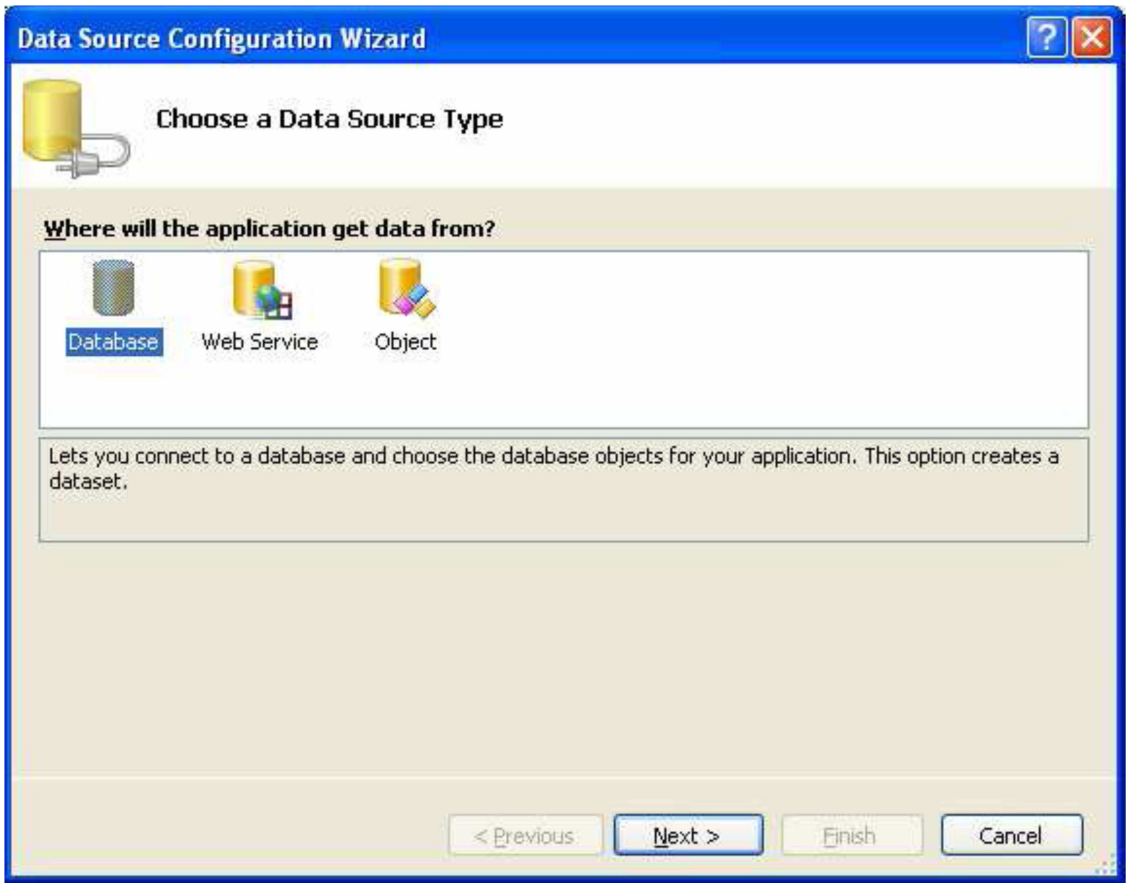
Access to the Database (this is one of the sections students mentioned it lacked in the previous section)

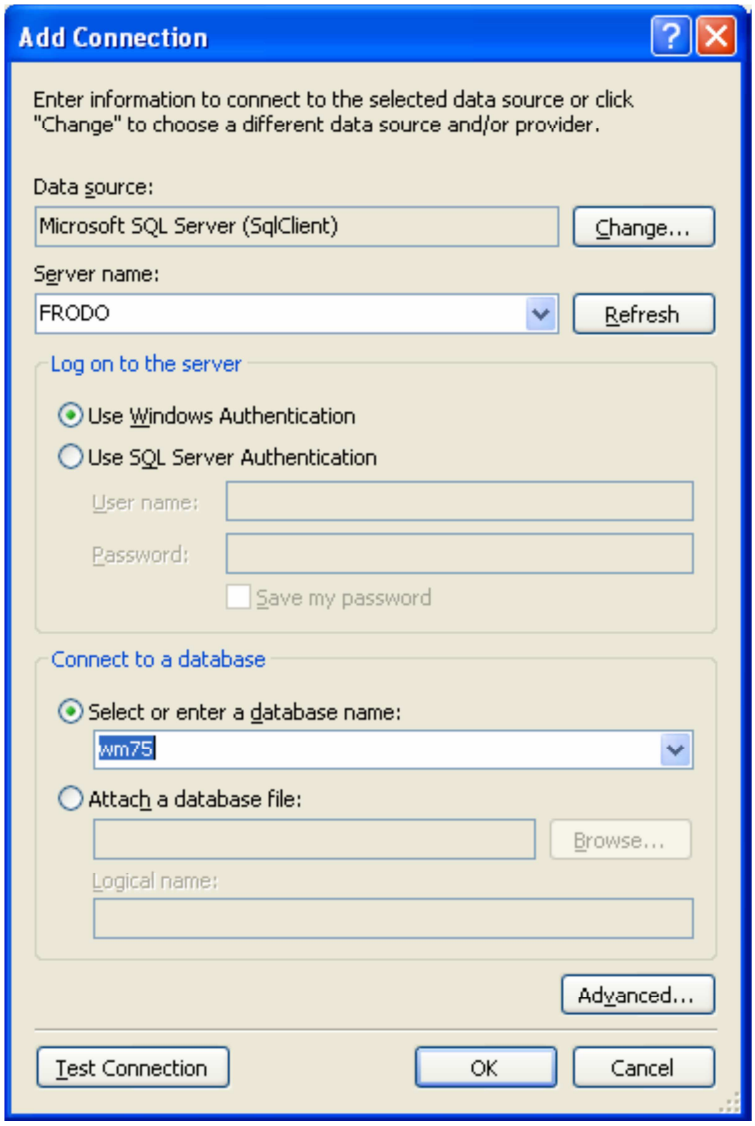
• In order to be able to access the database created it is necessary to add it as a data source

• Make sure SQL Server is running and your database is accessible

• Select Add New Data Source from the Data menu

• Then select Database





• On the next screen click on the New Connection button and set your connection details

– Select your Server

– Select your Database

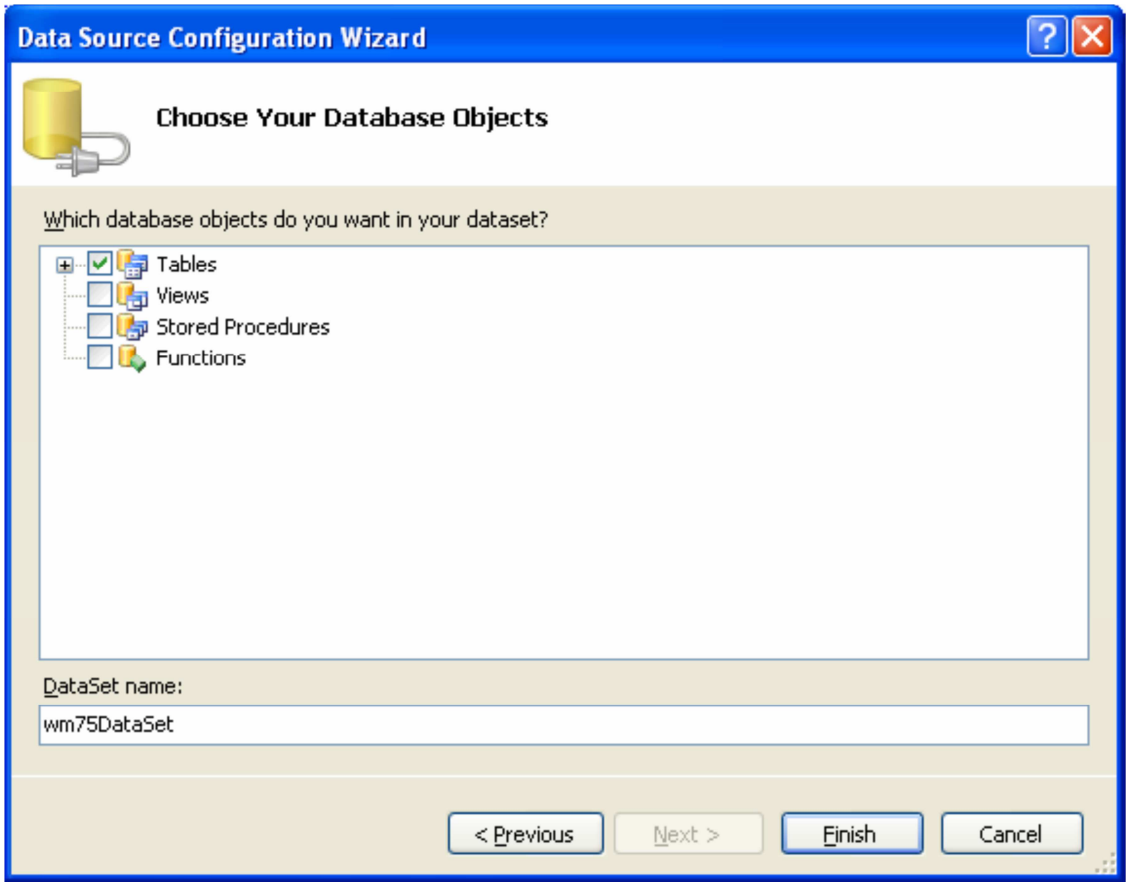
• Once set, test your connection and proceed to the next screen

• Make sure the checkbox is ticked

• Name your connection ConnectionString

• Tick the tables checkbox and then click finish

• Now you have a connection to the db and the connection string was created in the Settings.settings file



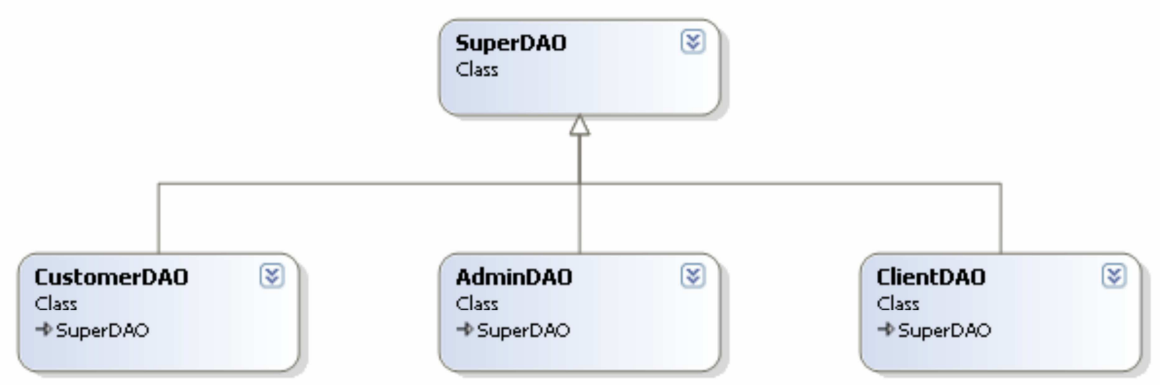
DAO - Reminder

• Data Access Objects provide abstract interfaces to data sources

• DAOs provide a clear separation between our business and persistence logic

• The DAOs will contain all the SQL code for reading and writing to the database

DAO – UML Structure



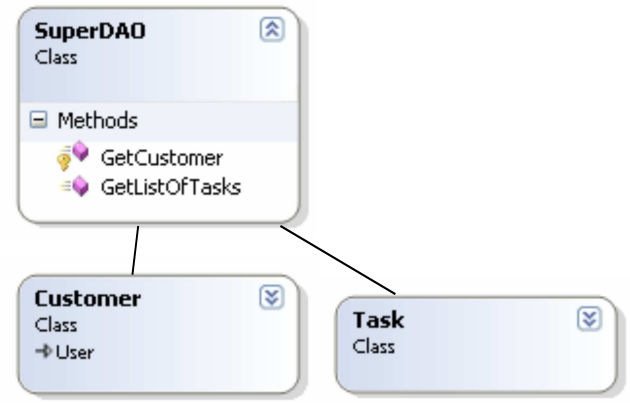
Class SuperDAO

• This is the base class DAO

• Contains 2 methods providing behaviour shared by the other DAOs

– GetCustomer

– GetListOfTasks



SuperDAO - GetCustomer method

```
protected Customer GetCustomer(int custId)
{
    string sql;
    SqlConnection cn;
    SqlCommand cmd;
    SqlDataReader dr;
    Customer cust;

    sql = "SELECT * FROM Customer WHERE CustomerId = " + custId;
    cn = new SqlConnection(Properties.Settings.Default.ConnectionString);
    cmd = new SqlCommand(sql, cn);

    try
    {
        cn.Open();
        dr = cmd.ExecuteReader(CommandBehavior.SingleRow);
        dr.Read();
        cust = new Customer(dr["Name"].ToString(), (int)dr["CustomerId"]);
        dr.Close();
    }
    catch (SqlException ex)
    {
        throw new Exception("Error Getting Customer", ex);
    }
    finally
    {
        cn.Close();
    }
    return cust;
}
```

- Declare objects necessary to access the DB
- SQL statement to retrieve customer details
- Create connection using the ConnectionString
- The SQL command is set to return a single row
- An instance cust of Customer is created
- The method returns cust

SuperDAO - GetListOfTasks method

```
public List<Task> GetListOfTasks(Guid projectId)
{
    string sql;
    SqlConnection cn;
    SqlCommand cmd;
    SqlDataReader dr;
    List<Task> tasks;
    tasks = new List<Task>();

    sql = "SELECT * FROM Task WHERE ProjectId = '" + projectId + "'";
    cn = new SqlConnection(Properties.Settings.Default.ConnectionString);
    cmd = new SqlCommand(sql, cn);

    try
    {
        cn.Open();
        dr = cmd.ExecuteReader();
        while (dr.Read())
        {
            Task t = new Task((Guid)dr["TaskId"], dr["Name"].ToString(), (Status)((int)dr["StatusId"]));
            tasks.Add(t);
        }
        dr.Close();
    }
    catch (SqlException ex)
    {
        throw new Exception("Error getting taks list", ex);
    }
    finally
    {
        cn.Close();
    }
    return tasks;
}
```

Class CustomerDAO

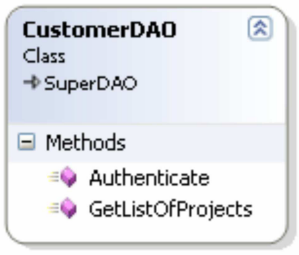
• This DAO provides DB access methods specific for the customer role

• Inherits from the SuperDAO class

• Two methods

– Authenticate

– GetListOfProjects



CustomerDAO – Authenticate method

```
public int Authenticate(string username, string password)
{
    string sql;
    SqlConnection cn;
    SqlCommand cmd;
    SqlDataReader dr;

    sql = String.Format("SELECT CustomerId FROM Customer WHERE Username='{0}' AND Password='{1}'", username, password);

    cn = new SqlConnection(Properties.Settings.Default.ConnectionString);
    cmd = new SqlCommand(sql, cn);
    int id = 0;
    try
    {
        cn.Open();
        dr = cmd.ExecuteReader(CommandBehavior.SingleRow);
        if (dr.Read())
        {
            id = (int)dr["CustomerId"];
        }
        dr.Close();
    }
    catch (SqlException ex)
    {
        throw new Exception("Error Accessing Database", ex);
    }
    finally
    {
        cn.Close();
    }
    return id;
}
```

CustomerDAO – GetListOfProjects method

```
try
{
    cn.Open();
    dr = cmd.ExecuteReader();
    while (dr.Read())
    {
        List<Task> tasks = new List<Task>();
        sql = "SELECT * FROM Task WHERE ProjectId = '" + dr["ProjectId"].ToString() + "'";
        cn2 = new SqlConnection(Properties.Settings.Default.ConnectionString);
        cmd2 = new SqlCommand(sql, cn2);
        cn2.Open();
        dr2 = cmd2.ExecuteReader();
        while (dr2.Read())
        {
            Task t = new Task((Guid)dr2["TaskId"], dr2["Name"].ToString(), (Status)dr2["StatusId"]);
            tasks.Add(t);
        }
        dr2.Close();
        Project p = new Project(dr["Name"].ToString(), (DateTime)dr["ExpectedStartDate"],
                                (DateTime)dr["ExpectedEndDate"], (Guid)dr["ProjectId"], tasks);
        projects.Add(p);
    }
    dr.Close();
}
catch (SqlException ex)
{
    throw new Exception("Error Getting list", ex);
}
finally
{
    cn.Close();
}
```

Class ClientDAO*

• Similar to CustomerDAO

• This DAO provides DB access methods specific for the TeamLeader role

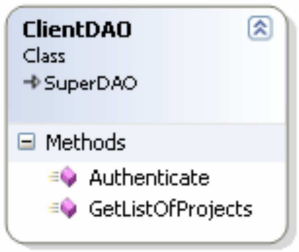
• Inherits from the SuperDAO class

• Two methods

– Authenticate

– GetListOfProjects

• This class provides the DB access reuired by the Java client



ClientDAO – Things to note

• SQL statement for Authenticate method

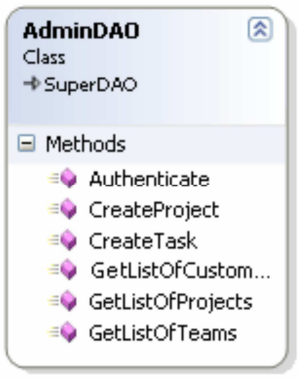
```
sql = String.Format("SELECT DISTINCT Person.Name, UserId, TeamId FROM Person INNER JOIN Team ON
(Team.TeamLeaderId = Person.UserId) WHERE Username='{0}' AND Password='{1}'",
username, password);
```

• GetListOfProjects method now returns all projects for a particular team, not for a particular customer which was the case in
CustomerDAO

```
public List<Project> GetListOfProjects(int teamId)
```

Class AdminDAO

• This DAO provides DB access methods specific for the Administrator role
• Inherits from the SuperDAO class
• Six methods
– Authenticate
– CreateProject
– CreateTask
– GetListOfCustomers
– GetListOfProjects
– GetListOfTeams



AdminDAO – CreateProject method

```
public void CreateProject(string name, DateTime startDate, DateTime endDate, int customerId, int administratorId)
{
    string sql;
    SqlConnection cn;
    SqlCommand cmd;

    Guid projectId = Guid.NewGuid();

    sql = "INSERT INTO Project (ProjectId, Name, ExpectedStartDate, ExpectedEndDate, CustomerId, AdministratorId)";
    sql += String.Format("VALUES ( '{0}', '{1}', '{2}', '{3}', {4}, {5})", projectId, name,
                         startDate, endDate, customerId, administratorId);
    cn = new SqlConnection(Properties.Settings.Default.ConnectionString);
    cmd = new SqlCommand(sql, cn);

    try
    {
        cn.Open();
        cmd.ExecuteNonQuery();

    }
    catch (SqlException ex)
    {
        throw new Exception("Error Inserting", ex);
    }
    finally
    {
        cn.Close();
    }

}
```

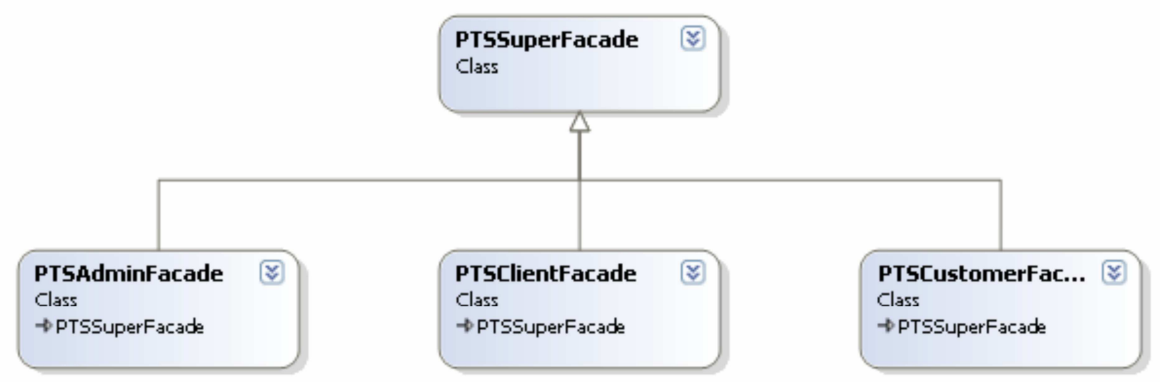
AdminDAO – Things to note

- SQL statement for Authenticate method ensures that only administrators can authenticate
- CreateTask method inserts a new task in the DB
- GetListOfCustomers returns all customers existing in the DB
- GetListOfProjects returns only the projects created by a particular administrator
- GetListOfTeams - returns all teams existing in the DB

Façade Objects

- Provide a publicly available interface to our business component

• One façade class for each type of user



Class PTSSuperFacade

• This is the base façade class

• Contains one methods providing behaviour shared by the other façades

– GetListOfTasks

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace PTSLibrary
6  {
7      class PTSSuperFacade
8      {
9          protected DAO.SuperDAO dao;
10
11         public PTSSuperFacade(DAO.SuperDAO dao)
12         {
13             this.dao = dao;
14         }
15
16         public Task[] GetListOfTasks(Guid projectId)
17         {
18             return (dao.GetListOfTasks(projectId)).ToArray();
19         }
20     }
21 }
```

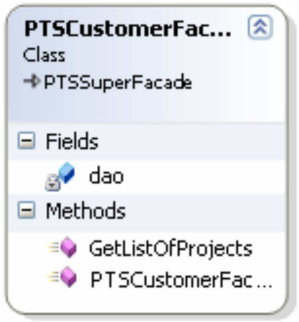
Class PTSCustomerFacade

• This facade provides a public interface for the customer web service

• Inherits from the PTSSuperFacade class

• One method

– GetListOfProjects



```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace PTSLibrary
6  {
7      class PTSCustomerFacade : PTSSuperFacade
8      {
9          private DAO.CustomerDAO dao;
10
11         public PTSCustomerFacade() : base(new DAO.CustomerDAO())
12         {
13             dao = (DAO.CustomerDAO)base.dao;
14         }
15
16         public Project[] GetListOfProjects(int customerId)
17         {
18             return (dao.GetListOfProjects(customerId)).ToArray();
19         }
20     }
21 }
22
```

Class PTSClientFacade

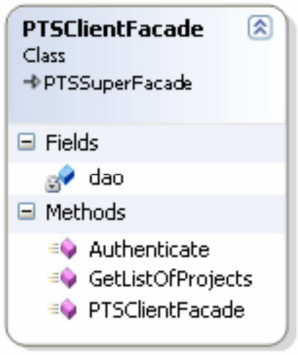
• This facade provides a public interface for the client web service used by the Java Client

• Inherits from the PTSSuperFacade class

• Two methods

– GetListOfProjects

– Authenticate



```
class PTSClientFacade : PTSSuperFacade
{
    private DAO.ClientDAO dao;

    public PTSClientFacade()
        : base(new DAO.ClientDAO())
    {
        dao = (DAO.ClientDAO)base.dao;
    }

    public TeamLeader Authenticate(string username, string password)
    {
        if (username == "" || password == "")
        {
            throw new Exception("Missing Data");
        }
        return dao.Authenticate(username, password);
    }

    public Project[] GetListOfProjects(int teamId)
    {
        return (dao.GetListOfProjects(teamId)).ToArray();
    }

}
```

Class PTSClientFacade

• This facade provides a public interface for the Administrator remote client

• Inherits from the PTSSuperFacade class

• Methods

– Authenticate

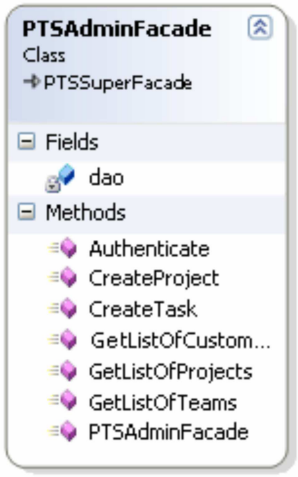
– CreateProject

– CreateTask

– GetListOfCustomers

– GetListOfProjects

– GetListOfTeams




```
class PTSClientFacade : PTSSuperFacade
{
    private DAO.ClientDAO dao;

    public PTSClientFacade()
        : base(new DAO.ClientDAO())
    {
        dao = (DAO.ClientDAO)base.dao;
    }

    public TeamLeader Authenticate(string username, string password)
    {
        if (username == "" || password == "")
        {
            throw new Exception("Missing Data");
        }
        return dao.Authenticate(username, password);
    }

    public Project[] GetListOfProjects(int teamId)
    {
        return (dao.GetListOfProjects(teamId)).ToArray();
    }

}
```

Summary

• This concludes the work on the PTSLibrary business component

• You should try to build the project by selecting Build PTSLibrary from the Build menu and fix any compilation errors that you might get

• A lot of code was written which you weren't able to test

– This is what you will be doing in the next session