Practical Lecture 2 Building a Business Component

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

2

Overview

- By now you should have:
  - Familiarised yourself with the requirements of the system
  - Developed the database for the system
- In this lecture we will build a business component which encapsulates the business functionality of the system

3

Learning Objectives

- Understand n-tier architectures
- Understand the use of components
- Create a business component in .NET using C#, which interacts with a database

4

Introduction

- In this practical session we 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

5

N-Tier Architectures

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

6

N-Tier Architectures /2

- 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

7

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

8

Business Tier /2

- The business tier will be defined using classes
- The collection (library) of classes representing our business tier will be deployed as a component
  - In our case a DLL

9

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)

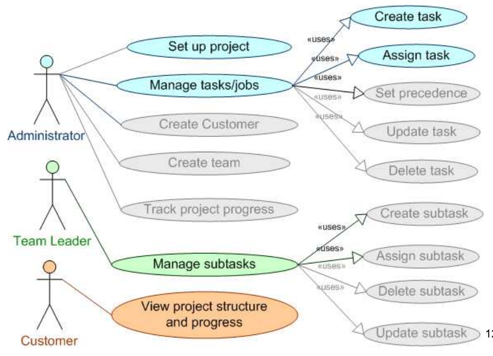
10

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 = **P**roject **T**racking **S**ystem)

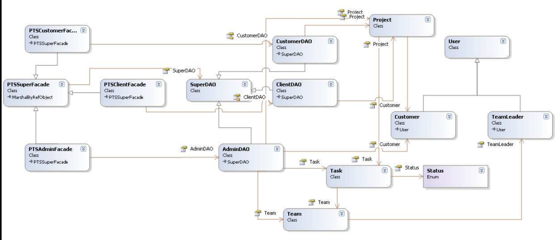
11

Use-Case Diagram



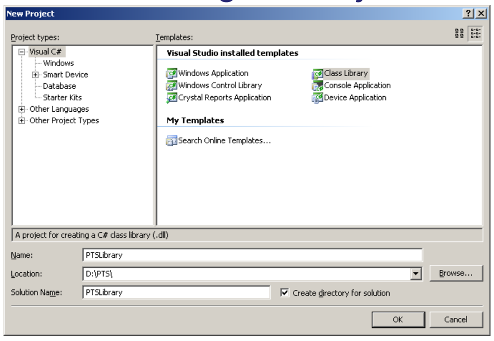
12

Class Diagram



13

Creating the Project



New Project

Project types:
- Visual C#
  - Windows
  - Smart Device
  - Database
  - Starter Kits
- Other Languages
- Other Project Types

Templates:

Visual Studio installed templates

Windows Application	Class Library
Windows Control Library	Console Application
Crystal Reports Application	Device Application

My Templates

Search Online Templates...

A project for creating a C# class library (.dll)

Name: PTSLibrary

Location: D:\PTS\

Solution Name: PTSLibrary

[✓] Create directory for solution

14

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

15

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.

16

Business Objects /2

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

17

Business Objects /3

- 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”

18

DAOs

- 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

19

DAOs /2

- 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)

20

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

21

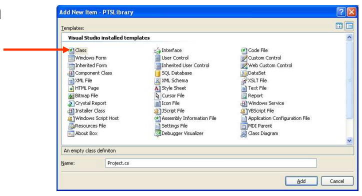
Façade Objects /2

- 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)

22

Creating classes

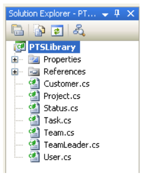
- Delete the Class1.cs file created by default when you created the new project
- Now create all the business classes
- Make sure that you select *Class* as the template for each



23

Creating Classes /2

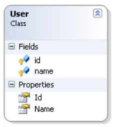
- You Solution Explorer should now look like this
- Each of the classes created only contains some default import statements (*using* statements), namespace declaration and class declaration. Lets add our desired state and behaviour
- Remember that we will only implement a subset of the functionality required to demonstrate the use of the system



24

Class User

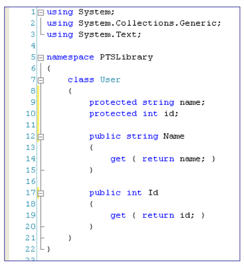
- This class represents a general user of the system
- It is the super (base) class for all more specialised classes representing users
  - Customer
  - TeamLeader
  - TeamMember
- The above three inherit from User



25

Class User /2

- 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; }
        }
    }
}
```

26

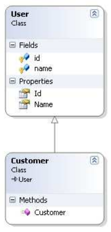
Class Customer

- 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

27

Class User /2

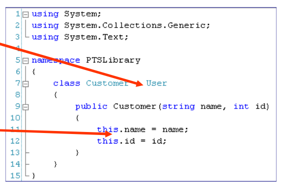
- All the code we need to write is
  - make the Customer inherit from User
  - add a constructor taking a name and id



28

Class Customer /3

- Customer inherits from User
- Q: What is the *this.name* referring to if *name* is not declared?



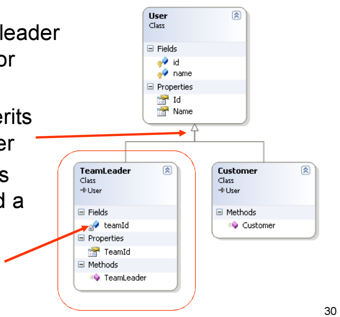
```
using System;
using System.Collections.Generic;
using System.Text;

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

29

Class TeamLeader

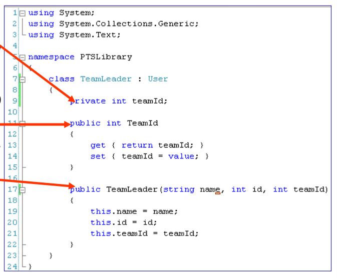
- Represents a leader of an internal or external team
- The class inherits from class User
- Similar to class Customer, add a constructor
- One new field



30

Class TeamLeader /2

- The new field
- Create a property *TeamId* to provide access to *teamId*
- Class constructor with 3 parameters, including one for *teamId*

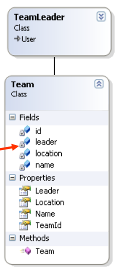


```
using System;
using System.Collections.Generic;
using System.Text;

namespace PTSLibrary
{
    class TeamLeader : User
    {
        private int teamId;

        public int TeamId
        {
            get { return teamId; }
            set { teamId = value; }
        }

        public TeamLeader(string name, int id, int teamId)
        {
            this.name = name;
            this.id = id;
            this.teamId = teamId;
        }
    }
}
```

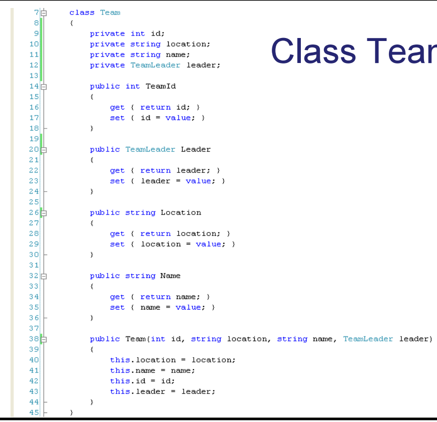
31

Class Team /1

- Team represents an internal or external team working for Out of Bounds Ltd
- It is linked to TeamLeader through association
- Other fields: id, location, name



32



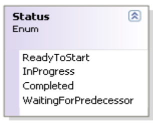
```
    class Team
    {
        private int id;
        private string location;
        private string name;
        private TeamLeader leader;

        public int TeamId
        {
            get { return id; }
            set { id = value; }
        }

        public TeamLeader Leader
        {
            get { return leader; }
            set { leader = value; }
        }

        public string Location
        {
            get { return location; }
            set { location = value; }
        }

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

        public Team(int id, string location, string name, TeamLeader leader)
        {
            this.location = location;
            this.name = name;
            this.id = id;
            this.leader = leader;
        }
    }
```

Class Team /2

33

Enum Status /1

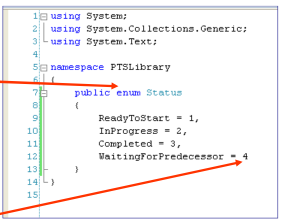
- Enum allows you to create a distinct value type
- Contains a set of named constants
- Can be converted to an integer
- Which is easier to read?
  - if(currentStatus == 3)
  - If(currentStatus == Status.Completed)



34

Enum Status /2

- 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

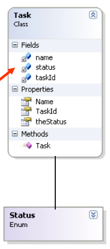


```
using System;
using System.Collections.Generic;
using System.Text;

namespace PTSLibrary
{
    public enum Status
    {
        ReadyToStart = 1,
        InProgress = 2,
        Completed = 3,
        WaitingForPredecessor = 4
    }
}
```

35

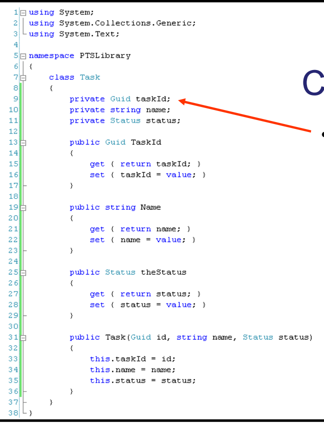
Class Task /1

- 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



36

Class Task /2

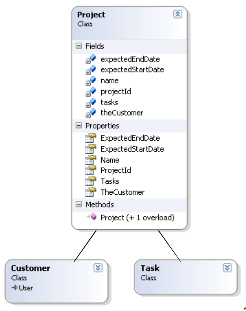


```
using System;
using System.Collections.Generic;
using System.Text;

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 /1

- 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



Class Project /2

- Note the data types used in declaring the variables
- *tasks* is declared using generic programming
- Generics:
  - Allow the creation of type-safe 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;
```

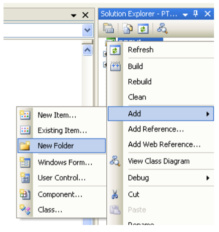
Class Project /3

- Two constructors (one setting the customer, one the tasks)
- For complete code of the class see the notes

```
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;
}
```

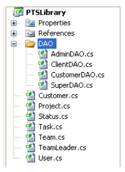
Creating DAOs

- 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*



Creating DAOs /2

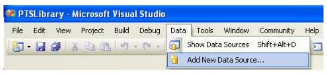
- Create 4 new classes in the DAO folder called:
  - SuperDAO
  - AdminDAO
  - CustomerDAO
  - ClientDAO
- These classes will have code to work with our database
  - To have access to the required classes we need to import namespaces System.Data and System.Data.SqlClient in each DAO class



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

Access to the Database

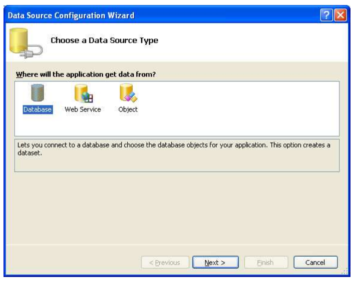
- In order to be able to access the database created it is necessary to add it as a data source
- Make sure SQL Server is running and your database is accessible
- Select *Add New Data Source* from the *Data* menu


PTSLibrary - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools Window Community Help
Show Data Sources Shift+Alt+D
Add New Data Source...

43

Access to the Database /2

- Then select Database

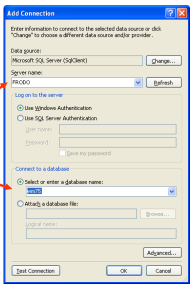

Data Source Configuration Wizard
Choose a Data Source Type
Where will the application get data from?
Database
Web Service
Object
< Previous
Next >
Finish
Cancel

Lets you connect to a database and choose the database objects for your application. This option creates a dataset.

44

Access to the Database /3

- 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



Enter information to connect to the selected data source or click "Change" to choose a different data source and/or provider.

Data source: Microsoft SQL Server (SqlClient)

Server name: FRODO

[✓ Use Windows Authentication] Use SQL Server Authentication

User name:

Password:

Save my password

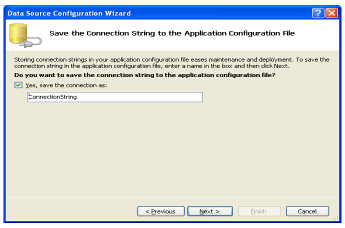
[✓ Select or enter a database name:] wm75

Attach a database file:

45

Access to the Database /4

- Make sure the checkbox is ticked
- Name your connection *ConnectionString*


Data Source Configuration Wizard
Save the Connection String to the Application Configuration File
< Previous
Next >
Finish
Cancel

Storing connection strings in your application configuration file eases maintenance and deployment. To save the connection string in the application configuration file, enter a name in the box and then click Next.

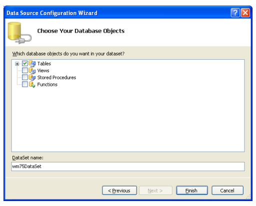
Do you want to save the connection string to the application configuration file?

✓ Yes, save the connection as: ConnectionString

46

Access to the Database /5

- 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


Data Source Configuration Wizard
Choose Your Database Objects
< Previous
Next >
Finish
Cancel

Which database objects do you want in your dataset?

✓ Tables

Views

Stored Procedures

Functions

DataSet name: wm75DataSet

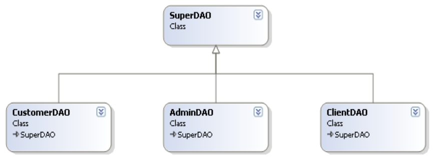
47

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

48

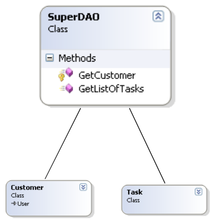
DAO – UML Structure



49

Class SuperDAO

- This is the base class DAO
- Contains 2 methods providing behaviour shared by the other DAOs
  - GetCustomer
  - GetListOfTasks



50

SuperDAO - GetCustomer method

- 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*

```
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;
}
```

51

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;
}
```

Note the use of single quotes: projectId is of type GUID and not a number

Returns possibly more than one row

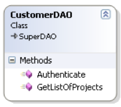
Iterates through all returned rows

Task constructor expects a Guid and Status

52

Class CustomerDAO /1

- This DAO provides DB access methods specific for the customer role
- Inherits from the SuperDAO class
- Two methods
  - Authenticate
  - GetListOfProjects



53

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;
}
```

Format item {0} is replaced with the value the username

Ensures that we only read from the SQL DataReader only if a matching CustomerId is returned. If not, 0 is returned

54

CustomerDAO – GetListOfProjects method

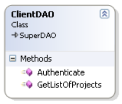
```
try
{
    cn.Open();
    dr = cmd.ExecuteReader();
    while (dr.Read())
    {
        List<Task> tasks = new List<Task>();
        sql = "SELECT * FROM Task WHERE ProjectId = '" + dr["ProjectId"].ToString() + "'";
        cn2 = new SqlConnection(Properties.Settings.Default.ConnectionString);
        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();
}
```

Two sets of DB related objects are created :
1. To retrieve projects
2. To retrieve all tasks for each project

55

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



56

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);

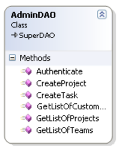
- GetListOfProjects method now returns all projects for a particular team, not for a particular customer which was the case in CustomerDAO

public List<Project> GetListOfProjects(int teamId)

57

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



58

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();
    }

}
```

Generating a new Guid

Executing an INSERT rather than a SELECT statement

59

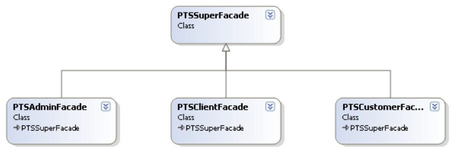
AdminDAO – Things to note

- 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

60

Façade Objects

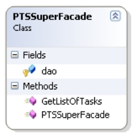
- Provide a publicly available interface to our business component
- One façade class for each type of user



61

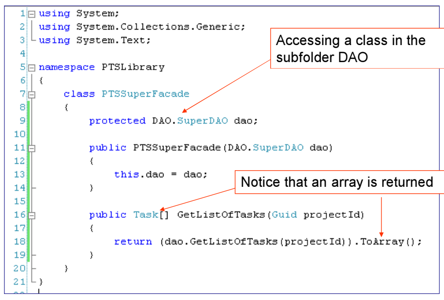
Class PTSSuperFacade

- This is the base façade class
- Contains one methods providing behaviour shared by the other façades
  - GetListOfTasks



62

Class PTSSuperFacade /2



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

namespace PTSLibrary
{
    class PTSSuperFacade
    {
        protected DAO.SuperDAO dao;

        public PTSSuperFacade(DAO.SuperDAO dao)
        {
            this.dao = dao;
        }

        public Task[] GetListOfTasks(Guid projectId)
        {
            return (dao.GetListOfTasks(projectId)).ToArray();
        }
    }
}
```

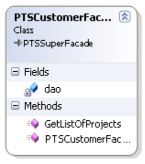
Accessing a class in the subfolder DAO

Notice that an array is returned

63

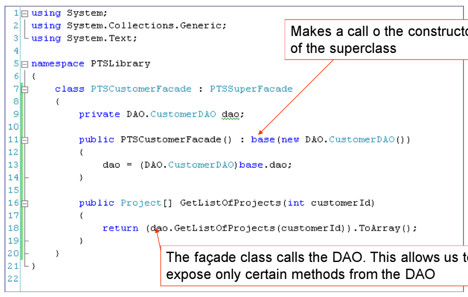
Class PTSCustomerFacade

- This facade provides a public interface for the customer web service
- Inherits from the PTSSuperFacade class
- One method
  - GetListOfProjects



64

Class PTSCustomerFacade /2



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

namespace PTSLibrary
{
    class PTSCustomerFacade : PTSSuperFacade
    {
        private DAO.CustomerDAO dao;

        public PTSCustomerFacade() : base(new DAO.CustomerDAO())
        {
            dao = (DAO.CustomerDAO)base.dao;
        }

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

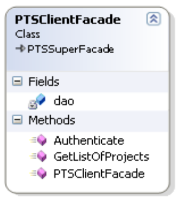
Makes a call o the constructor of the superclass

The façade class calls the DAO. This allows us to expose only certain methods from the DAO

65

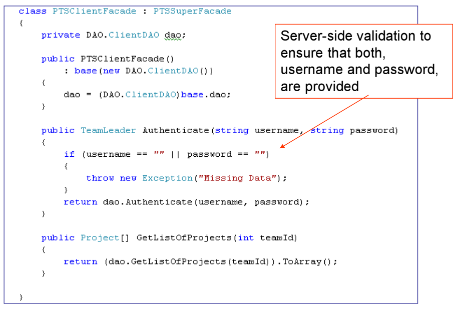
Class PTSClientFacade

- 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



66

Class PTSClientFacade /2



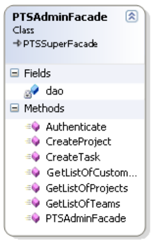
```
class PTSClientFacade : PTSSuperFacade
{
    private DAO.ClientDAO dao;

    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();
    }

}
```

Server-side validation to ensure that both, username and password, are provided

67

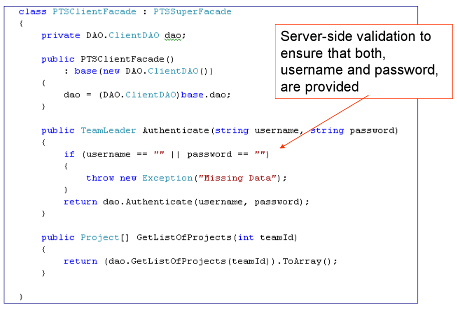
Class PTSClientFacade

- This facade provides a public interface for the Administrator remote client
- Inherits from the PTSSuperFacade class
- Methods
  - Authenticate
  - CreateProject
  - CreateTask
  - GetListOfCustomers
  - GetListOfProjects
  - GetListOfTeams



68

Class PTSClientFacade /2



```
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();
    }

}
```

Server-side validation to ensure that both, username and password, are provided

69

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

70