**Factory Design Pattern Introduction**

**we will discuss**

1. What is Factory Design Pattern
2. Implementation Guidelines
3. Simple factory implementation

**What is Factory Design Pattern**  
  
**Gang of Four Definition**  
“Define an interface for creating an object, but let sub-classes decide which class to instantiate. The Factory method lets a class defer instantiation it uses to sub-classes”  
  
  
  
Factory pattern is one of the most used design patterns in real world applications  
  
Factory pattern creates object without exposing the creation logic to the client and refer to newly created object using a common interface  
  
  
  
From the above diagram, client uses factory and creates the product.  
  
**Implementation Guidelines :**We need to choose Factory Pattern when

* The Object needs to be extended to subclasses
* The Classes doesn’t know what exact sub-classes it has to create
* The Product implementation tend to change over time and the Client remains unchanged

**Simple Factory Example :**Business Requirement  
  
Differentiate employees as permanent and contract and segregate their pay scales as well as bonus based on their employee types  
  
We can address the above requirement with the below implementations

1. Implement without Factory Pattern
2. Use a Simple Factory
3. Enhance Simple factory to Factory Method Pattern

We will be working on the Employee Portal that we used in the Singleton tutorials. Please refer to them before proceeding.  
  
**Prerequisite steps**  
**Step 1 :** Enhance the DB model to add **Employee\_Type**Table

CREATE TABLE [dbo].[Employee\_Type]

(

    [Id]           INT           IDENTITY (1, 1) NOT NULL,

    [EmployeeType] VARCHAR (150) NOT NULL,

    PRIMARY KEY CLUSTERED ([Id] ASC)

)

**Step 2 :**Add Permanent and Contract Employees as Master Data  
  
**Step 3 :** Add new columns EmployeeTypeID, Bonus, HourlyPay to Emplyee Table and add Foreign key constraint to the Emp

CREATE TABLE [dbo].[Employee]

(

    [Id]             INT          IDENTITY (1, 1) NOT NULL,

    [Name]           VARCHAR (50) NOT NULL,

    [JobDescription] VARCHAR (50) NOT NULL,

    [Number]         VARCHAR (50) NOT NULL,

    [Department]     VARCHAR (50) NOT NULL,

    [HourlyPay]      DECIMAL (18) NOT NULL,

    [Bonus]          DECIMAL (18) NOT NULL,

    [EmployeeTypeID] INT          NOT NULL,

    PRIMARY KEY CLUSTERED ([Id] ASC),

    CONSTRAINT [FK\_Employee\_EmployeeType] FOREIGN KEY ([EmployeeTypeID]) REFERENCES [dbo].[Employee\_Type] ([Id])

)

**Step 4 :** Update the Emloyee Model edmx file with the latest changes  
  
**Step 5 :** Create new BaseController and Move the Singleton Exception logic  to the base controller

    public class BaseController : Controller

    {

        private ILog \_ILog;

        public BaseController()

        {

            \_ILog = Log.GetInstance;

        }

        protected override void OnException(ExceptionContext filterContext)

        {

            \_ILog.LogException(filterContext.Exception.ToString());

            filterContext.ExceptionHandled = true;

            this.View("Error").ExecuteResult(this.ControllerContext);

        }

    }

**Step 6 :** Regenerate the EmployeesController and its corresponding views  
  
**Step 7 :** Comment the code in Create and Update views which accepts inputs for Bonus and HourlyPay  
  
**Solution 1: Implement without Factory Pattern**  
  
**EmployeeController.cs**

        [HttpPost]

        [ValidateAntiForgeryToken]

        public ActionResult Create([Bind(Include = "Id,Name,JobDescription,Number,Department,HourlyPay,Bonus,EmployeeTypeID")] Employee employee)

        {

            if (ModelState.IsValid)

            {

**if (employee.EmployeeTypeID == 1)**

**{**

**employee.HourlyPay = 8;**

**employee.Bonus = 10;**

**}**

**else if (employee.EmployeeTypeID == 2)**

**{**

**employee.HourlyPay = 12;**

**employee.Bonus = 5;**

**}**

                db.Employees.Add(employee);

                db.SaveChanges();

                return RedirectToAction("Index");

            }

            ViewBag.EmployeeTypeID = new SelectList(db.Employee\_Type, "Id", "EmployeeType", employee.EmployeeTypeID);

            return View(employee);

        }

**The above code introduces**

* Tight coupling between Controller class and Business logic
* For any new employee type addition, we end up modifying the controller code adding extra over heads in the development and testing process

Using a simple factory eliminates the above drawbacks.

**Solution 2: Implement with Simple Factory**

**Step 1 :**Add new Manager folder and add the below interface and classes

**IEmployeeManager.cs**

    public interface IEmployeeManager

    {

        decimal GetBonus();

        decimal GetPay();

    }

**ContractEmployeeManager.cs**

    public class ContractEmployeeManager : IEmployeeManager

    {

        public decimal GetBonus()

        {

            return 5;

        }

        public decimal GetPay()

        {

            return 12;

        }

    }

**PermanentEmployeeManager.cs**

    public class PermanentEmployeeManager : IEmployeeManager

    {

        public decimal GetBonus()

        {

            return 10;

        }

        public decimal GetPay()

        {

            return 8;

        }

    }

**Step 2 :**Create Factory folder and add the below Manager class

**EmployeeManagerFactory.cs**

    public class EmployeeManagerFactory

    {

        public IEmployeeManager GetEmployeeManager(int employeeTypeID)

        {

            IEmployeeManager returnValue = null;

            if (employeeTypeID == 1)

            {

                returnValue = new PermanentEmployeeManager();

            }

            else if (employeeTypeID == 2)

            {

                returnValue = new ContractEmployeeManager();

            }

            return returnValue;

        }

    }

**Step 3 :**Update the employee’s controller to consume the factory.

 [HttpPost]

        [ValidateAntiForgeryToken]

        public ActionResult Create([Bind(Include = "Id,Name,JobDescription,Number,Department,HourlyPay,Bonus,EmployeeTypeID")] Employee employee)

        {

            if (ModelState.IsValid)

            {

                EmployeeManagerFactory empFactory = new EmployeeManagerFactory();

                IEmployeeManager empManager = empFactory.GetEmployeeManager(employee.EmployeeTypeID);

                employee.Bonus = empManager.GetBonus();

                employee.HourlyPay = empManager.GetPay();

                db.Employees.Add(employee);

                db.SaveChanges();

                return RedirectToAction("Index");

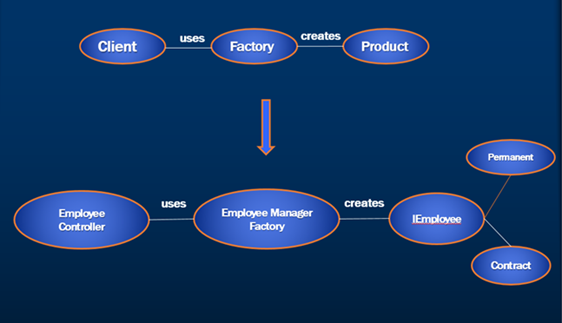
            }

            ViewBag.EmployeeTypeID = new SelectList(db.Employee\_Type, "Id", "EmployeeType", employee.EmployeeTypeID);

            return View(employee);

        }

**Simple factory implementation is illustrated below**



### Factory Method Design Pattern

**we will learn**

1. Simple Factory
2. Factory Method Pattern Implementation

**Recap Simple Factory**

1. Simple factory abstracts the creation details of the product
2. Simple factory refers to the newly created object through an interface
3. Any new type creation is handled with a change of code in the factory class and not in the client code

public class EmployeeManagerFactory

{

    public IEmployeeManager GetEmployeeManager(int employeeTypeID)

    {

        IEmployeeManager returnValue = null;

        if (employeeTypeID == 1)

        {

            returnValue = new PermanentEmployeeManager();

        }

        else if (employeeTypeID == 2)

        {

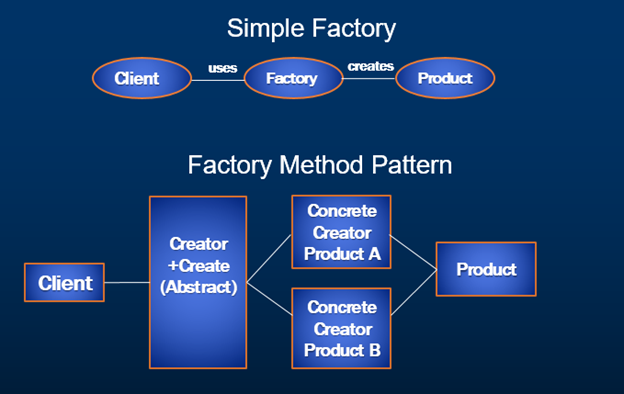
            returnValue = new ContractEmployeeManager();

        }

        return returnValue;

    }

}

**Factory Representation**  
  
  
From the above diagram, Factory representation block in the simple factory is replaced with Abstract Creator which further creates Concrete Creator A and Concrete Creator B Products providing another level of abstraction.  
  
**Factory Method Pattern Example**  
  
**Business Requirement**

1. Differentiate employees as permanent and contract and segregate their pay scales as well as bonus based on their employee types.  ( We have achieved this using [simple factory in Part 8](https://www.youtube.com/watch?v=bGqw8crGZ7Y) of the [Design Patterns tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhUbctIoxoVTrklN3LMwTCmd))
2. Calculate Permanent employee house rent allowance
3. Calculate Contract employee medical allowance

**Steps to solve the above business requirement**

**Step 1:**Add HouseAllowance and MedicalAllowance to the existing Employee table.

CREATE TABLE [dbo].[Employee] (

    [Id]               INT          IDENTITY (1, 1) NOT NULL,

    [Name]             VARCHAR (50) NOT NULL,

    [JobDescription]   VARCHAR (50) NOT NULL,

    [Number]           VARCHAR (50) NOT NULL,

    [Department]       VARCHAR (50) NOT NULL,

    [HourlyPay]        DECIMAL (18) NOT NULL,

    [Bonus]            DECIMAL (18) NOT NULL,

    [EmployeeTypeID]   INT          NOT NULL,

**[HouseAllowance]   DECIMAL (18) NULL,**

**[MedicalAllowance] DECIMAL (18) NULL,**

    PRIMARY KEY CLUSTERED ([Id] ASC),

    CONSTRAINT [FK\_Employee\_EmployeeType] FOREIGN KEY ([EmployeeTypeID]) REFERENCES [dbo].[Employee\_Type] ([Id]) );

**Step 2:**Open EmployeePortal.edmx under the Models folder of the solution and update the model from the database (Right click on the model designer and choose update from database option)

**Step 3:**Create FactoryMethod folder under existing Factory folder and add BaseEmployeeFactory class.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using Web.Managers;

using Web.Models;

namespace Web.Factory.FactoryMethod

{

    public abstract class BaseEmployeeFactory

    {

        protected Employee \_emp;

        public BaseEmployeeFactory(Employee emp)

        {

            \_emp = emp;

        }

        public Employee ApplySalary()

        {

            IEmployeeManager manager = this.Create();

            \_emp.Bonus = manager.GetBonus();

            \_emp.HourlyPay = manager.GetPay();

            return \_emp;

        }

        public abstract IEmployeeManager Create();

    }

}

**Step 4:**Create ContractEmployeeFactory class under FactoryMethod folder.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using Web.Managers;

using Web.Models;

namespace Web.Factory.FactoryMethod

{

    public class ContractEmployeeFactory : BaseEmployeeFactory

    {

        public ContractEmployeeFactory(Employee emp) : base(emp)

        {

        }

        public override IEmployeeManager Create()

        {

            ContractEmployeeManager manager = new ContractEmployeeManager();

            \_emp.MedicalAllowance = manager.GetMedicalAllowance();

            return manager;

        }

    }

}

**Step 5:**Create PermanentEmployeeFactory class under FactoryMethod folder.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using Web.Managers;

using Web.Models;

namespace Web.Factory.FactoryMethod

{

    public class PermanentEmployeeFactory : BaseEmployeeFactory

    {

        public PermanentEmployeeFactory(Employee emp) : base(emp)

        {

        }

        public override IEmployeeManager Create()

        {

            PermanentEmployeeManager manager = new PermanentEmployeeManager();

            \_emp.HouseAllowance = manager.GetHouseAllowance();

            return manager;

        }

    }

}

**Step 6:**Create EmployeeManagerFactory class under FactoryMethod folder and add new Method CreateFactory which returns BaseEmployeeFactory.

CreateFactory method is responsible to return base factory which is the base class of Permanent and Contract Factories.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using Web.Models;

namespace Web.Factory.FactoryMethod

{

    public class EmployeeManagerFactory

    {

        public BaseEmployeeFactory CreateFactory(Employee emp)

        {

            BaseEmployeeFactory returnValue = null;

            if(emp.EmployeeTypeID ==1)

            {

                returnValue = new PermanentEmployeeFactory(emp);

            }

            else if (emp.EmployeeTypeID ==2)

            {

                returnValue = new ContractEmployeeFactory(emp);

            }

            return returnValue;

        }

    }

}

**Step 7:**Now, integrate the FactoryMethod in the EmployeesController’s Create method and replace the existing logic of simple factory method as stated below.

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Create([Bind(Include = "Id,Name,JobDescription,Number,Department,HourlyPay,Bonus,EmployeeTypeID")] Employee employee)

{

    if (ModelState.IsValid)

    {

        BaseEmployeeFactory empFactory =

                    new EmployeeManagerFactory().CreateFactory(employee);

        empFactory.ApplySalary();

        db.Employees.Add(employee);

        db.SaveChanges();

        return RedirectToAction("Index");

    }

    ViewBag.EmployeeTypeID = new SelectList(db.Employee\_Type, "Id",

        "EmployeeType", employee.EmployeeTypeID);

    return View(employee);

}

**Step 8:**Enhance the current Employee index view to add Medical Allowance and House Allowance columns.

**Step 9:**Run the application and notice that we have achieved the business requirement by using Factory Method pattern. Below illustrated diagram depicts how we have converted the simple to factory method implementation.

