Singleton pattern

Singleton е създаващ шаблон за дизайн, който се използва в обектно-ориентираното програмиране. Този шаблон се използва обикновено в моделирането на обекти, които трябва да бъдат глобално достъпни за обектите на приложението (например обекта съдрържайки структурите с настройките на програмата ) или обекти, които се нуждаят от максимално късна инициализация за пестенето на ресурси от паметта.

**class** Singleton

{

**private** **static** Singleton \_instance;

**protected** Singleton()

{

}

**public** **static** Singleton Instance()

{

**if** (\_instance == **null**)

{

\_instance = new Singleton();

}

**return** \_instance;

}

}

Proxy pattern

Предоставя заместител (с минимална функционалност) или обвивка за даден обект с цел контрол на достъпа до него.

Remote Proxy предоставя обект за локален достъп “имитация” на реален отдалечен обект.

Virtual Proxy – прост обект замества по-сложен и предоставя същия интерфейс сложния обект се създава само при нужда.

Protection Proxy контролира достъпа до основния обект

Предимства:

Скриване на конкретните класове продукти от клиента

Възможност за пазене на логиката на отдалечена машина (сървър)

Възможност за ускорение на основните процеси в дадено приложение

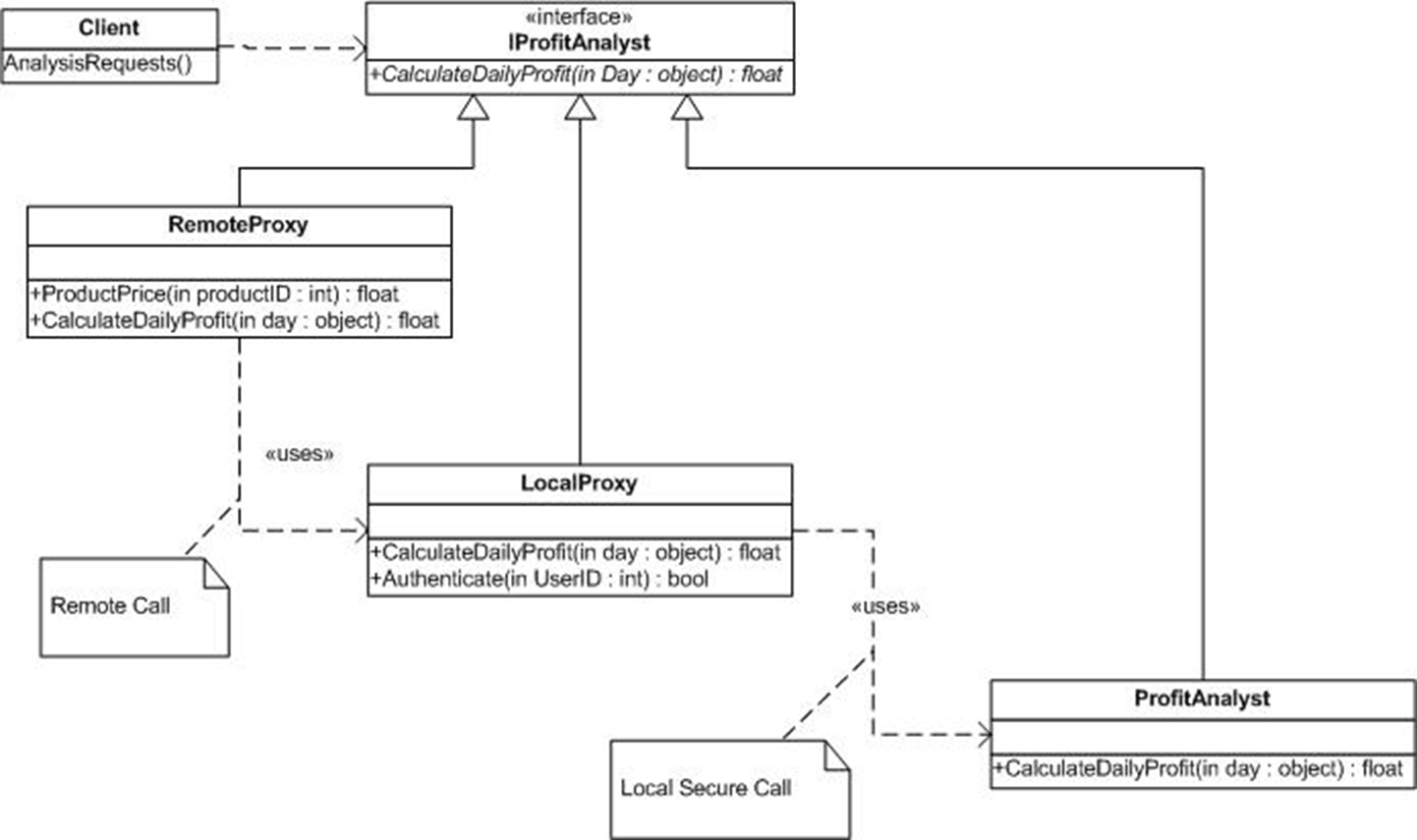
Лесно добавяне и премахване на обекти по време на изпълнение\

Недостатъци:

Всички класове трябва да имплементират един и същ интерфейс

Увеличено количество код за по-сложните задачи

Усложняване на архитектурата на приложението, което изисква по-добра запознатост с нея



**public interface IProfitAnalyst**

**{**

**float CalculateDailyProfit(DateTime day);**

**}**

**public class RemoteProxy: IProfitAnalyst**

**{**

**. . .**

**public float ProductPrice(int productID)**

**{**

**return (float) Convert.ToDouble(mPrices[productID]);**

**}**

**public float CalculateDailyProfit(DateTime day)**

**{**

**LocalProxy localProxy = new LocalProxy();**

**if (localProxy.Authenticate(mUserID))**

**{**

**return localProxy.CalculateDailyProfit(day);**

**}**

**else {return 0;}**

**}**

**}**

**public class LocalProxy: IProfitAnalyst**

**{**

**private float LoadDailyProfit(DateTime day)**

**{**

**return (float) 5000.00 + day.DayOfYear;**

**}**

**public bool Authenticate(int userID)**

**{**

**return (userID > 0);**

**}**

**}**

**public class ProfitAnalystReal: IProfitAnalyst**

**{**

**. . .**

**public float CalculateDailyProfit(DateTime Day)**

**{**

**//Data retrieval from the OLTP database is done**

**//many calculations that take a lot of time are**

**//done here**

**//let's assume the result returned is 100000.00**

**return (float) 100000.00;**

**}**

**}**

Adapter pattern

Преобразува интерфейса на даден клас в друг такъв, очакван от клиента

Дава възможност на класове да работят заедно в случаите, когато това е невъзможно поради несъвместими интерфейси

Adapter

**abstract class Shape**

**{**

**public abstract void Draw();**

**}**

**class Line: Shape**

**{**

**public override void Draw()**

**{**

**Console.WriteLine("Line.Draw()");**

**}**

**}**

**class Spline**

**{**

**public void Draw()**

**{**

**Console.WriteLine("Spline.Draw()");**

**}**

**}**

**class SplineAdapter: Shape**

**{**

**private Spline splineLine = new Spline();**

**public override void Draw()**

**{**

**Console.WriteLine("Inside**

**SplineAdapter.Draw");**

**splineLine.Draw();**

**}**

**}**

**static void Main(string[] args)**

**{**

**ArrayList shapeHolder = new ArrayList();**

**Line line = new Line();**

**shapeHolder.Add(line);**

**SplineAdapter spline = new SplineAdapter();**

**shapeHolder.Add(spline);**

**foreach (Shape shape in shapeHolder)**

**{**

**shape.Draw();**

**}**

**}**

**Резултат:**

**Line.Draw()**

**Inside SplineAdapter.Draw**

**Spline.Draw()**