# 15. Design Patterns

## Singleton Design Pattern

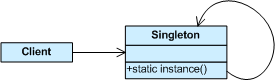
Източник: http://sourcemaking.com/design\_patterns/singleton

* Осигурява една единствена инстанция от даден клас.
* Инстанцията не се създава, преди да е необходимо използването й, т.е. преди да се извика за първи път. (lazy initialization)

Прави се клас, който отговаря за създаването, иниициализирането и достъпа до даден клас. Работи в private интанция, като осигурява публичен интерфейс, който да енкапсулира детайлите по създаването й, и осигурява интерфейс за достъп до въпросната иснтанция.

Този дизайн има смисъл да се използва, ако е трудно по друг начин да се даде глобален достъп до нея, ако е необходима lazy initialization и ако инстанцията ще се ползва от много места.

Има следната структура:



Пример:

|  |
| --- |
| using System;  class MainApp  {    static void Main()  {  // Constructor is protected -- cannot use new  Singleton s1 = Singleton.Instance();  Singleton s2 = Singleton.Instance();  if (s1 == s2)  {  Console.WriteLine("Objects are the same instance");  }  // Wait for user  Console.Read();  }  }  // "Singleton"  class Singleton  {  private static Singleton instance;  // Note: Constructor is 'protected'  protected Singleton()  {  }  public static Singleton Instance()  {  // Use 'Lazy initialization'  if (instance == null)  {  instance = new Singleton();  }  return instance;  }  }  } |

## Adapter Design Pattern

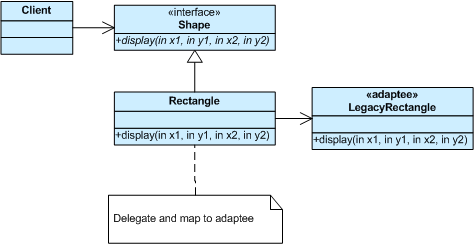
Източник: http://sourcemaking.com/design\_patterns/adapter

Осигурява нов интерфейс за даден клас, какъв клиента очаква. По този начин се осигурява съвместната работа на класове, която иначе би била невъзможна, поради несъвместимост в интерфейсите.

* Дава възможност да се работи със стари компоненти в нова система.

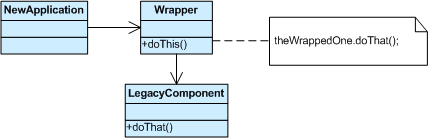
Основно се използва, когато има някаква функционалност, която трябва да се преизползва, но не е съвместима с архитектурата на разработваната система.

Структурата може да бъде следната:



В този пример, стария(legacy) метод Rectangle, изисква координатите на горния ъгъл и широчина и височина за да се извика метода Display(). Клиента обаче иска да ползва същия метод чрез интерфейс, който изисква координатите на два срещуположни ъгъла. Adapter-а се грижи това да е възможно.

Схемата в по-общ вариант:



Пример:

|  |
| --- |
| using System;  class MainApp  {  static void Main()  {  // Create adapter and place a request  Target target = new Adapter();  target.Request();  // Wait for user  Console.Read();  }  }  // "Target"  class Target  {  public virtual void Request()  {  Console.WriteLine("Called Target Request()");  }  }  // "Adapter"  class Adapter : Target  {  private Adaptee adaptee = new Adaptee();  public override void Request()  {  // Possibly do some other work  // and then call SpecificRequest  adaptee.SpecificRequest();  }  }  // "Adaptee"  class Adaptee  {  public void SpecificRequest()  {  Console.WriteLine("Called SpecificRequest()");  }  } |

## Memento Design Pattern

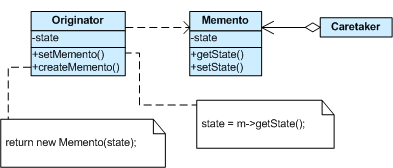
Източник: <http://sourcemaking.com/design_patterns/memento>

* Цели, без да се нарушава енкапсулацията, да се извлече вътрешното състояние на обект, така че до да може да бъде възстановено.
* Предоставя “undo” или “rollback” до някакво установено състояние.

Използва се, в случай когато даден обект трябва да може да се върне в определено състояние.  
В този дизайн има три основни роли:

* Originator – обект, който знае как да се запази.
* Caretaker – обект, който знае кога и защо Originator-а трябва да се запази или да се зареди.
* Memento – обекта (шаблон), в който се пише и чете от Originator и използва от Caretaker.

Има следната структура:



Пример:

|  |
| --- |
| using System;  namespace DoFactory.GangOfFour.Memento.Structural  {  // MainApp startup class for Structural  // Memento Design Pattern.  class MainApp  {  // Entry point into console application.  static void Main()  {  Originator o = new Originator();  o.State = "On";  // Store internal state  Caretaker c = new Caretaker();  c.Memento = o.CreateMemento();  // Continue changing originator  o.State = "Off";  // Restore saved state  o.SetMemento(c.Memento);  // Wait for user  Console.ReadKey();  }  }  // The 'Originator' class  class Originator  {  private string \_state;  // Property  public string State  {  get { return \_state; }  set  {  \_state = value;  Console.WriteLine("State = " + \_state);  }  }  // Creates memento  public Memento CreateMemento()  {  return (new Memento(\_state));  }  // Restores original state  public void SetMemento(Memento memento)  {  Console.WriteLine("Restoring state...");  State = memento.State;  }  }  // The 'Memento' class  class Memento  {  private string \_state;  // Constructor  public Memento(string state) {  this.\_state = state;  }  // Gets or sets state  public string State  {  get { return \_state; }  }  }  // The 'Caretaker' class  class Caretaker  {  private Memento \_memento;  // Gets or sets memento  public Memento Memento  {  set { \_memento = value; }  get { return \_memento; }  }  }  } |