**Homework assignment –** Select 3 design patterns

* Write a short description (about half page) for each of them (prefer Bulgarian language)
* Describe their motivation, intent, applicability, known uses, implementation, consequences, structure, related patterns, etc.
* Provide C# examples for their use
* Provide a UML diagram or image of the pattern
* You can download it from the Internet

Превеждането от английски беше истинско мъчение. Моля за извинение, ако няма много смисъл на места, тъй като до преди академията не съм се занимавал с програмиране и терминологията на български ми е безкрайно непозната.

1. **Memento pattern.** 
   1. [По-подробна информация (на английски език).](http://www.oodesign.com/memento-pattern.html)
   2. Кратко описание:

Без да се нарушава капсулирането на данни се улавя състоянието на даден обект, така че обектът да може да бъде възстановен до това състояние по-късно.

Участващите елементи в този шаблон са три:

* „първоизточник“ (Originator) – Създава „спомен“, съдържащ моментна снимка на текущото вътрешно състояние на даден обект. По-късно използва този „спомен“, за да възстанови състоянието на обекта.
* „спомен“ (Memento) – Запазва вътрешното състояние на обекта „първоизточник“. Спомена може да запазва различни части от състоянието на първоизточника, според нуждите му. Има два интерфейса – един за „уредника“, който трябва да не нарушава капсулирането на данните, както и един за „първоизточника“, който трябва да осигурява достъп до всички променливи, характеризиращи състоянието на обекта, нужни за да може да се възстанови предишното състояние.
* „уредник“ (CareTaker) – Носи отговорност за съхраняването на „спомена“, като не извършва никакви операции върху съдържанието му.
  1. Известни примери за използването му:
* Операции за undo и restore в повечето софтуерни продукти.
* Транзакции на информация, получавана от и изпращана към бази данни.
  1. Прост пример за използването му:

namespace Memento

{

using System;

  class Main

  {

    static void Main()

    {

      Originator o = new Originator();

      o.State = "On";

      Caretaker c = new Caretaker();

      // Съхранява състоянието на обекта първоизточник ‘о’

c.Memento = o.CreateMemento();

      // Променя състоянието на обекта първоизточник ‘о’

      o.State = "Off";

      // Възстановяваме към запаметеното състояние

      o.SetMemento(c.Memento);

    }

  }

  class Originator

  {

    private string \_state;

    // Свойство, което ще се съхранява чрез Memento

    public string State

    {

      get { return \_state; }

      set

      {

        \_state = value;

        Console.WriteLine("State = " + \_state);

      }

    }

    // Създава „спомена“

    public Memento CreateMemento()

    {

      return (new Memento(\_state));

    }

    // Възстановява оригиналното състояние

    public void SetMemento(Memento memento)

    {

      Console.WriteLine("Restoring state...");

      State = memento.State;

    }

  }

  class Memento

  {

    private string \_state;

    public Memento(string state)

    {

this.\_state = state;

}

    public string State

    {

      get { return \_state; }

    }

  }

  class Caretaker

  {

    private Memento \_memento;

    public Memento Memento

    {

      set { \_memento = value; }

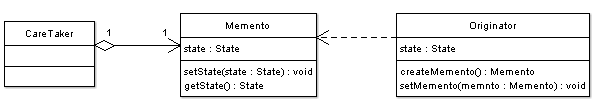
      get { return \_memento; }

    }

  }

}

* 1. UML диаграма:



1. **Command pattern.**
   1. [По-подробна информация (на английски език).](http://www.oodesign.com/command-pattern.html)
   2. Кратко описание:

Капсулира различни заявки като обекти, като по този начин може да се параметризират клиенти с различни заявки, да се подреждат заявките за последователно изпълнение (в опашки /queue/).

Участващите елементи в този шаблон са пет:

* „команда“ (Command) – Декларира интерфейс за изпълнение на операция.
* „конкретна команда“ (ConcreteCommand) – Разширява интерфейса на командата, имплементирайки метод за изпълнение (Execute()) чрез позоваване на съответните операции в приемника. Тя дефинира връзката между приемника и конкретното действие.
* „клиент“ (Client) – Създава обект ConcreteCommand и задава приемника ѝ.
* „повикващ“ (Invoker) – подава на командата искане да се изпълни.
* „приемник“ (Receiver) – знае как да изпълни дадена операция.

Клиентът (Client) поисква да бъде изпълнена дадена команда. Повикващия обект (Invoker) поема командата, капсулира я и я подава за изпълнение (записва я в опашка /queue/, ако има други команди за изпълнение преди конкретната). Конкретната команда (ConcreteCommand), от която зависи повиканата команда изпраща резултата на приемника (Receiver).

* 1. Известни примери за използването му:
* Операции за undo и redo.
* Асинхронно повикване на методи.
  1. Прост пример за използването му:

namespace Command

{

using System;

  class Main

  {

    static void Main()

    {

      Receiver receiver = new Receiver();

      Command command = new ConcreteCommand(receiver);

      Invoker invoker = new Invoker();

      invoker.SetCommand(command);

      invoker.ExecuteCommand();

    }

  }

  abstract class Command

  {

    protected Receiver receiver;

    public Command(Receiver receiver)

    {

      this.receiver = receiver;

    }

    public abstract void Execute();

  }

  class ConcreteCommand : Command

  {

    public ConcreteCommand(Receiver receiver) : base(receiver) { }

    public override void Execute()

    {

      receiver.Action();

    }

  }

  class Receiver

  {

    public void Action()

    {

      Console.WriteLine("Called Receiver.Action()");

    }

  }

  class Invoker

  {

    private Command \_command;

    public void SetCommand(Command command)

    {

      this.\_command = command;

    }

    public void ExecuteCommand()

    {

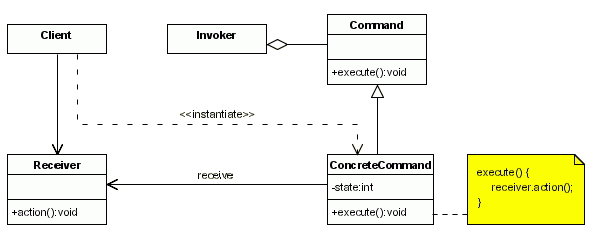
      \_command.Execute();

    }

  }

}

* 1. UML диаграма:



1. **Strategy pattern.** 
   1. [По-подробна информация (на английски език).](http://www.oodesign.com/strategy-pattern.html)
   2. Кратко описание:

Често се случват ситуации, при които дадени класове се различават единствено по поведението си. За тези случаи е добра идея да се изолират алгоритмите в отделни класове, за да се осигури възможността да се избират различни алгоритми по време на изпълнение.

Целта е да се дефинира семейство от алгоритми, всяко от което капсулирано, и да се направят взаимно заменяеми. Този шаблон за дизайн позволява на алгоритмите да са зависими единствено от клиента, който ги използва.

Участващите елементи в този шаблон са условно три:

* „стратегия“ (Strategy) – Дефинира общ интерфейс за всички поддържани алгоритми. Контекста използва този интерфейс, за да повиква алгоритъма, дефиниран от конкретна стратегия.
* „конкретна стратегия“ (ConcreteStrategy) – Всяка конкретна стратегия имплементира даден алгоритъм.
* „контекст“ (Context) – Съдържа референция към обект Strategy. Може да дефинира интерфейс, който да позволява на стратегията да достъпва данните му.

Контекста съдържа референция към конкретната стратегия, която трябва да бъде използвана. Когато се извика дадена операция се включва алгоритъма от обекта Strategy. Контекста не се интересува от имплементацията на стратегията. При необходимост може да се дефинират допълнителни обекти, които да подават данни от контекста към стратегията.

* 1. Прост пример за използването му:

namespace Strategy

{

using System;

class Main

  {

    static void Main()

    {

      Context context;

      // Три контекста, следващи различни стратегии

      context = new Context(new ConcreteStrategyA());

      context.ContextInterface();

      context = new Context(new ConcreteStrategyB());

      context.ContextInterface();

    context = new Context(new ConcreteStrategyC());

      context.ContextInterface();

   }

  }

 abstract class Strategy

 {

  public abstract void AlgorithmInterface();

}

  class ConcreteStrategyA : Strategy

 {

   public override void AlgorithmInterface()

   {

     Console.WriteLine("Called ConcreteStrategyA.AlgorithmInterface()");

  }

  }

 class ConcreteStrategyB : Strategy

{

  public override void AlgorithmInterface()

  {

    Console.WriteLine("Called ConcreteStrategyB.AlgorithmInterface()");

   }

  }

 class ConcreteStrategyC : Strategy

 {

   public override void AlgorithmInterface()

   {

     Console.WriteLine("Called ConcreteStrategyC.AlgorithmInterface()");

   }

}

class Context

 {

    private Strategy \_strategy;

   public Context(Strategy strategy)

   {

    this.\_strategy = strategy;

    }

   public void ContextInterface()

   {

     \_strategy.AlgorithmInterface();

   }

}

}

* 1. UML диаграма:

